File: ./target/aarch64-apple-darwin/release/build/libsqlite3-sys-3b51a

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
```

```
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
```

```
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
```

```
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
```

```
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
```

```
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
```

```
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
```

```
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
```

```
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
```

```
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
```

```
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xWrite: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *const ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
```

```
pub xTruncate: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xDeviceCharacteristics: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xShmMap: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        iPg: ::std::os::raw::c_int,
        pgsz: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
```

```
            arg3: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmLock: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            offset: ::std::os::raw::c_int,
            n: ::std::os::raw::c_int,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
```

```
pub pNext: *mut sqlite3_vfs,
pub zName: *const ::std::os::raw::c_char,
pub pAppData: *mut ::std::os::raw::c_void,
pub xOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: sqlite3_filename,
        arg2: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
        pOutFlags: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xDelete: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        syncDir: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xAccess: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        flags: ::std::os::raw::c_int,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFullPathname: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        nOut: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xDlOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zFilename: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
```

```
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
>,
pub xRandomness: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSleep: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        microseconds: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTime: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
>,
pub xGetLastError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTimeInt64: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xSetSystemCall: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        arg2: sqlite3_syscall_ptr,
    ) -> ::std::os::raw::c_int,
>,
pub xGetSystemCall: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
```

```
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
```

```
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
```

```
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
```

```
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
```

```
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
```

```
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
```

```
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
```

```
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
```

```
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
```

```
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
```

```
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
```

```
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
```

```
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
```

```
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
```

```
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
```

```
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
```

```
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
```

```
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
```

```
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
```

```
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
```

```
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xConnect: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xBestIndex: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: *mut sqlite3_index_info,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDisconnect: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xDestroy: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            ppCursor: *mut *mut sqlite3_vtab_cursor,
        ) -> ::std::os::raw::c_int,
    >,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
    >,
    pub xFilter: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vtab_cursor,
            idxNum: ::std::os::raw::c_int,
            idxStr: *const ::std::os::raw::c_char,
            argc: ::std::os::raw::c_int,
            argv: *mut *mut sqlite3_value,
        ) -> ::std::os::raw::c_int,
    >,
```

```
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        nArg: ::std::os::raw::c_int,
        zName: *const ::std::os::raw::c_char,
        pxFunc: *mut ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        ppArg: *mut *mut ::std::os::raw::c_void,
```

```
        ) -> ::std::os::raw::c_int,
    >,
    pub xRename: ::std::option::Option<
        unsafe extern "C" fn(
            pVtab: *mut sqlite3_vtab,
            zNew: *const ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSavepoint: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
```

```
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
```

```
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
```

```
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
```

```
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
```

```
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
```

```
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
```

```
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
```

```
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
```

```
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
```

```
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
```

```
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
```

```
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
```

```
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
```

```
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
```

```
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
```

```
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
```

```
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
```

```
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
    pub xUserData: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
    >,
    pub xColumnCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xRowCount: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pnRow: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xColumnTotalSize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iCol: ::std::os::raw::c_int,
            pnToken: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            pCtx: *mut ::std::os::raw::c_void,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    arg1: *mut ::std::os::raw::c_void,
                    arg2: ::std::os::raw::c_int,
                    arg3: *const ::std::os::raw::c_char,
                    arg4: ::std::os::raw::c_int,
                    arg5: ::std::os::raw::c_int,
                    arg6: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xPhraseSize: ::std::option::Option<
        unsafe extern "C" fn(
```

```
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            arg1: *mut Fts5Context,
            pAux: *mut ::std::os::raw::c_void,
            xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetAuxdata: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            bClear: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xPhraseFirst: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
            arg4: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNext: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
            piOff: *mut ::std::os::raw::c_int,
        ),
    >,
    pub xPhraseFirstColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNextColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
```

```
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
```

```
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./target/aarch64-apple-darwin/release/build/typenum-5ef8f1658

```
/**
Type aliases for many constants.

This file is generated by typenum's build script.

For unsigned integers, the format is `U` followed by the number. We define

- Numbers 0 through 1024
- Powers of 2 below `u64::MAX`
- Powers of 10 below `u64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm};

# #[allow(dead_code)]
type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

For positive signed integers, the format is `P` followed by the number and
signed integers it is `N` followed by the number. For the signed integer ze
`Z0`. We define aliases for

- Numbers -1024 through 1024
- Powers of 2 between `i64::MIN` and `i64::MAX`
- Powers of 10 between `i64::MIN` and `i64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm, PInt, NInt};

# #[allow(dead_code)]
type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
# #[allow(dead_code)]
type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```
```

```
# Example
```rust
# #[allow(unused_imports)]
use typenum::{U0, U1, U2, U3, U4, U5, U6};
# #[allow(unused_imports)]
use typenum::{N3, N2, N1, Z0, P1, P2, P3};
# #[allow(unused_imports)]
use typenum::{U774, N17, N10000, P1024, P4096};
```

We also define the aliases `False` and `True` for `B0` and `B1`, respective
*/
#[allow(missing_docs)]
pub mod consts {
    use crate::uint::{UInt, UTerm};
    use crate::int::{PInt, NInt};

    pub use crate::bit::{B0, B1};
    pub use crate::int::Z0;

    pub type True = B1;
    pub type False = B0;
    pub type U0 = UTerm;
    pub type U1 = UInt<UTerm, B1>;
    pub type P1 = PInt<U1>; pub type N1 = NInt<U1>;
    pub type U2 = UInt<UInt<UTerm, B1>, B0>;
    pub type P2 = PInt<U2>; pub type N2 = NInt<U2>;
    pub type U3 = UInt<UInt<UTerm, B1>, B1>;
    pub type P3 = PInt<U3>; pub type N3 = NInt<U3>;
    pub type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    pub type P4 = PInt<U4>; pub type N4 = NInt<U4>;
    pub type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    pub type P5 = PInt<U5>; pub type N5 = NInt<U5>;
    pub type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
    pub type P6 = PInt<U6>; pub type N6 = NInt<U6>;
    pub type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
    pub type P7 = PInt<U7>; pub type N7 = NInt<U7>;
    pub type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
    pub type P8 = PInt<U8>; pub type N8 = NInt<U8>;
    pub type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;
    pub type P9 = PInt<U9>; pub type N9 = NInt<U9>;
    pub type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;
    pub type P10 = PInt<U10>; pub type N10 = NInt<U10>;
    pub type U11 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>;
    pub type P11 = PInt<U11>; pub type N11 = NInt<U11>;
    pub type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;
    pub type P12 = PInt<U12>; pub type N12 = NInt<U12>;
    pub type U13 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>;
    pub type P13 = PInt<U13>; pub type N13 = NInt<U13>;
    pub type U14 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>;
    pub type P14 = PInt<U14>; pub type N14 = NInt<U14>;
```

```
pub type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;
pub type P15 = PInt<U15>; pub type N15 = NInt<U15>;
pub type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
pub type P16 = PInt<U16>; pub type N16 = NInt<U16>;
pub type U17 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>;
pub type P17 = PInt<U17>; pub type N17 = NInt<U17>;
pub type U18 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B0>;
pub type P18 = PInt<U18>; pub type N18 = NInt<U18>;
pub type U19 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B1>;
pub type P19 = PInt<U19>; pub type N19 = NInt<U19>;
pub type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;
pub type P20 = PInt<U20>; pub type N20 = NInt<U20>;
pub type U21 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B1>;
pub type P21 = PInt<U21>; pub type N21 = NInt<U21>;
pub type U22 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B0>;
pub type P22 = PInt<U22>; pub type N22 = NInt<U22>;
pub type U23 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B1>;
pub type P23 = PInt<U23>; pub type N23 = NInt<U23>;
pub type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;
pub type P24 = PInt<U24>; pub type N24 = NInt<U24>;
pub type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
pub type P25 = PInt<U25>; pub type N25 = NInt<U25>;
pub type U26 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B0>;
pub type P26 = PInt<U26>; pub type N26 = NInt<U26>;
pub type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;
pub type P27 = PInt<U27>; pub type N27 = NInt<U27>;
pub type U28 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B0>;
pub type P28 = PInt<U28>; pub type N28 = NInt<U28>;
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
pub type U35 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P35 = PInt<U35>; pub type N35 = NInt<U35>;
pub type U36 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P36 = PInt<U36>; pub type N36 = NInt<U36>;
pub type U37 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P37 = PInt<U37>; pub type N37 = NInt<U37>;
pub type U38 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P38 = PInt<U38>; pub type N38 = NInt<U38>;
pub type U39 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P39 = PInt<U39>; pub type N39 = NInt<U39>;
pub type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P40 = PInt<U40>; pub type N40 = NInt<U40>;
```

```
pub type U41 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P41 = PInt<U41>; pub type N41 = NInt<U41>;
pub type U42 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P42 = PInt<U42>; pub type N42 = NInt<U42>;
pub type U43 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P43 = PInt<U43>; pub type N43 = NInt<U43>;
pub type U44 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P44 = PInt<U44>; pub type N44 = NInt<U44>;
pub type U45 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P45 = PInt<U45>; pub type N45 = NInt<U45>;
pub type U46 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P46 = PInt<U46>; pub type N46 = NInt<U46>;
pub type U47 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P47 = PInt<U47>; pub type N47 = NInt<U47>;
pub type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P48 = PInt<U48>; pub type N48 = NInt<U48>;
pub type U49 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P49 = PInt<U49>; pub type N49 = NInt<U49>;
pub type U50 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P50 = PInt<U50>; pub type N50 = NInt<U50>;
pub type U51 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P51 = PInt<U51>; pub type N51 = NInt<U51>;
pub type U52 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P52 = PInt<U52>; pub type N52 = NInt<U52>;
pub type U53 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P53 = PInt<U53>; pub type N53 = NInt<U53>;
pub type U54 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P54 = PInt<U54>; pub type N54 = NInt<U54>;
pub type U55 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P55 = PInt<U55>; pub type N55 = NInt<U55>;
pub type U56 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P56 = PInt<U56>; pub type N56 = NInt<U56>;
pub type U57 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P57 = PInt<U57>; pub type N57 = NInt<U57>;
pub type U58 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P58 = PInt<U58>; pub type N58 = NInt<U58>;
pub type U59 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P59 = PInt<U59>; pub type N59 = NInt<U59>;
pub type U60 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P60 = PInt<U60>; pub type N60 = NInt<U60>;
pub type U61 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P61 = PInt<U61>; pub type N61 = NInt<U61>;
pub type U62 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P62 = PInt<U62>; pub type N62 = NInt<U62>;
pub type U63 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P63 = PInt<U63>; pub type N63 = NInt<U63>;
pub type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P64 = PInt<U64>; pub type N64 = NInt<U64>;
pub type U65 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P65 = PInt<U65>; pub type N65 = NInt<U65>;
pub type U66 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P66 = PInt<U66>; pub type N66 = NInt<U66>;
```

```
pub type U67 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P67 = PInt<U67>; pub type N67 = NInt<U67>;
pub type U68 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P68 = PInt<U68>; pub type N68 = NInt<U68>;
pub type U69 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P69 = PInt<U69>; pub type N69 = NInt<U69>;
pub type U70 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P70 = PInt<U70>; pub type N70 = NInt<U70>;
pub type U71 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P71 = PInt<U71>; pub type N71 = NInt<U71>;
pub type U72 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P72 = PInt<U72>; pub type N72 = NInt<U72>;
pub type U73 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P73 = PInt<U73>; pub type N73 = NInt<U73>;
pub type U74 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P74 = PInt<U74>; pub type N74 = NInt<U74>;
pub type U75 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P75 = PInt<U75>; pub type N75 = NInt<U75>;
pub type U76 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P76 = PInt<U76>; pub type N76 = NInt<U76>;
pub type U77 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P77 = PInt<U77>; pub type N77 = NInt<U77>;
pub type U78 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P78 = PInt<U78>; pub type N78 = NInt<U78>;
pub type U79 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P79 = PInt<U79>; pub type N79 = NInt<U79>;
pub type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P80 = PInt<U80>; pub type N80 = NInt<U80>;
pub type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P81 = PInt<U81>; pub type N81 = NInt<U81>;
pub type U82 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P82 = PInt<U82>; pub type N82 = NInt<U82>;
pub type U83 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P83 = PInt<U83>; pub type N83 = NInt<U83>;
pub type U84 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P84 = PInt<U84>; pub type N84 = NInt<U84>;
pub type U85 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P85 = PInt<U85>; pub type N85 = NInt<U85>;
pub type U86 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P86 = PInt<U86>; pub type N86 = NInt<U86>;
pub type U87 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P87 = PInt<U87>; pub type N87 = NInt<U87>;
pub type U88 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P88 = PInt<U88>; pub type N88 = NInt<U88>;
pub type U89 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P89 = PInt<U89>; pub type N89 = NInt<U89>;
pub type U90 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P90 = PInt<U90>; pub type N90 = NInt<U90>;
pub type U91 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P91 = PInt<U91>; pub type N91 = NInt<U91>;
pub type U92 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P92 = PInt<U92>; pub type N92 = NInt<U92>;
```

```
pub type U93 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P93 = PInt<U93>; pub type N93 = NInt<U93>;
pub type U94 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P94 = PInt<U94>; pub type N94 = NInt<U94>;
pub type U95 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P95 = PInt<U95>; pub type N95 = NInt<U95>;
pub type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P96 = PInt<U96>; pub type N96 = NInt<U96>;
pub type U97 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P97 = PInt<U97>; pub type N97 = NInt<U97>;
pub type U98 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P98 = PInt<U98>; pub type N98 = NInt<U98>;
pub type U99 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P99 = PInt<U99>; pub type N99 = NInt<U99>;
pub type U100 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P100 = PInt<U100>; pub type N100 = NInt<U100>;
pub type U101 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P101 = PInt<U101>; pub type N101 = NInt<U101>;
pub type U102 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P102 = PInt<U102>; pub type N102 = NInt<U102>;
pub type U103 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P103 = PInt<U103>; pub type N103 = NInt<U103>;
pub type U104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P104 = PInt<U104>; pub type N104 = NInt<U104>;
pub type U105 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P105 = PInt<U105>; pub type N105 = NInt<U105>;
pub type U106 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P106 = PInt<U106>; pub type N106 = NInt<U106>;
pub type U107 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P107 = PInt<U107>; pub type N107 = NInt<U107>;
pub type U108 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P108 = PInt<U108>; pub type N108 = NInt<U108>;
pub type U109 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P109 = PInt<U109>; pub type N109 = NInt<U109>;
pub type U110 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P110 = PInt<U110>; pub type N110 = NInt<U110>;
pub type U111 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P111 = PInt<U111>; pub type N111 = NInt<U111>;
pub type U112 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P112 = PInt<U112>; pub type N112 = NInt<U112>;
pub type U113 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P113 = PInt<U113>; pub type N113 = NInt<U113>;
pub type U114 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P114 = PInt<U114>; pub type N114 = NInt<U114>;
pub type U115 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P115 = PInt<U115>; pub type N115 = NInt<U115>;
pub type U116 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P116 = PInt<U116>; pub type N116 = NInt<U116>;
pub type U117 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P117 = PInt<U117>; pub type N117 = NInt<U117>;
pub type U118 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P118 = PInt<U118>; pub type N118 = NInt<U118>;
```

```
pub type U119 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P119 = PInt<U119>; pub type N119 = NInt<U119>;
pub type U120 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P120 = PInt<U120>; pub type N120 = NInt<U120>;
pub type U121 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P121 = PInt<U121>; pub type N121 = NInt<U121>;
pub type U122 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P122 = PInt<U122>; pub type N122 = NInt<U122>;
pub type U123 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P123 = PInt<U123>; pub type N123 = NInt<U123>;
pub type U124 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P124 = PInt<U124>; pub type N124 = NInt<U124>;
pub type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P125 = PInt<U125>; pub type N125 = NInt<U125>;
pub type U126 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P126 = PInt<U126>; pub type N126 = NInt<U126>;
pub type U127 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P127 = PInt<U127>; pub type N127 = NInt<U127>;
pub type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P128 = PInt<U128>; pub type N128 = NInt<U128>;
pub type U129 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P129 = PInt<U129>; pub type N129 = NInt<U129>;
pub type U130 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P130 = PInt<U130>; pub type N130 = NInt<U130>;
pub type U131 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P131 = PInt<U131>; pub type N131 = NInt<U131>;
pub type U132 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P132 = PInt<U132>; pub type N132 = NInt<U132>;
pub type U133 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P133 = PInt<U133>; pub type N133 = NInt<U133>;
pub type U134 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P134 = PInt<U134>; pub type N134 = NInt<U134>;
pub type U135 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P135 = PInt<U135>; pub type N135 = NInt<U135>;
pub type U136 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P136 = PInt<U136>; pub type N136 = NInt<U136>;
pub type U137 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P137 = PInt<U137>; pub type N137 = NInt<U137>;
pub type U138 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P138 = PInt<U138>; pub type N138 = NInt<U138>;
pub type U139 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P139 = PInt<U139>; pub type N139 = NInt<U139>;
pub type U140 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P140 = PInt<U140>; pub type N140 = NInt<U140>;
pub type U141 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P141 = PInt<U141>; pub type N141 = NInt<U141>;
pub type U142 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P142 = PInt<U142>; pub type N142 = NInt<U142>;
pub type U143 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P143 = PInt<U143>; pub type N143 = NInt<U143>;
pub type U144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P144 = PInt<U144>; pub type N144 = NInt<U144>;
```

```
pub type U145 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P145 = PInt<U145>; pub type N145 = NInt<U145>;
pub type U146 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P146 = PInt<U146>; pub type N146 = NInt<U146>;
pub type U147 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P147 = PInt<U147>; pub type N147 = NInt<U147>;
pub type U148 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P148 = PInt<U148>; pub type N148 = NInt<U148>;
pub type U149 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P149 = PInt<U149>; pub type N149 = NInt<U149>;
pub type U150 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P150 = PInt<U150>; pub type N150 = NInt<U150>;
pub type U151 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P151 = PInt<U151>; pub type N151 = NInt<U151>;
pub type U152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P152 = PInt<U152>; pub type N152 = NInt<U152>;
pub type U153 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P153 = PInt<U153>; pub type N153 = NInt<U153>;
pub type U154 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P154 = PInt<U154>; pub type N154 = NInt<U154>;
pub type U155 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P155 = PInt<U155>; pub type N155 = NInt<U155>;
pub type U156 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P156 = PInt<U156>; pub type N156 = NInt<U156>;
pub type U157 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P157 = PInt<U157>; pub type N157 = NInt<U157>;
pub type U158 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P158 = PInt<U158>; pub type N158 = NInt<U158>;
pub type U159 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P159 = PInt<U159>; pub type N159 = NInt<U159>;
pub type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P160 = PInt<U160>; pub type N160 = NInt<U160>;
pub type U161 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P161 = PInt<U161>; pub type N161 = NInt<U161>;
pub type U162 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P162 = PInt<U162>; pub type N162 = NInt<U162>;
pub type U163 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P163 = PInt<U163>; pub type N163 = NInt<U163>;
pub type U164 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P164 = PInt<U164>; pub type N164 = NInt<U164>;
pub type U165 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P165 = PInt<U165>; pub type N165 = NInt<U165>;
pub type U166 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P166 = PInt<U166>; pub type N166 = NInt<U166>;
pub type U167 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P167 = PInt<U167>; pub type N167 = NInt<U167>;
pub type U168 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P168 = PInt<U168>; pub type N168 = NInt<U168>;
pub type U169 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P169 = PInt<U169>; pub type N169 = NInt<U169>;
pub type U170 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P170 = PInt<U170>; pub type N170 = NInt<U170>;
```

```
pub type U171 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P171 = PInt<U171>; pub type N171 = NInt<U171>;
pub type U172 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P172 = PInt<U172>; pub type N172 = NInt<U172>;
pub type U173 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P173 = PInt<U173>; pub type N173 = NInt<U173>;
pub type U174 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P174 = PInt<U174>; pub type N174 = NInt<U174>;
pub type U175 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P175 = PInt<U175>; pub type N175 = NInt<U175>;
pub type U176 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P176 = PInt<U176>; pub type N176 = NInt<U176>;
pub type U177 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P177 = PInt<U177>; pub type N177 = NInt<U177>;
pub type U178 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P178 = PInt<U178>; pub type N178 = NInt<U178>;
pub type U179 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P179 = PInt<U179>; pub type N179 = NInt<U179>;
pub type U180 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P180 = PInt<U180>; pub type N180 = NInt<U180>;
pub type U181 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P181 = PInt<U181>; pub type N181 = NInt<U181>;
pub type U182 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P182 = PInt<U182>; pub type N182 = NInt<U182>;
pub type U183 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P183 = PInt<U183>; pub type N183 = NInt<U183>;
pub type U184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P184 = PInt<U184>; pub type N184 = NInt<U184>;
pub type U185 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P185 = PInt<U185>; pub type N185 = NInt<U185>;
pub type U186 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P186 = PInt<U186>; pub type N186 = NInt<U186>;
pub type U187 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P187 = PInt<U187>; pub type N187 = NInt<U187>;
pub type U188 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P188 = PInt<U188>; pub type N188 = NInt<U188>;
pub type U189 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P189 = PInt<U189>; pub type N189 = NInt<U189>;
pub type U190 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P190 = PInt<U190>; pub type N190 = NInt<U190>;
pub type U191 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P191 = PInt<U191>; pub type N191 = NInt<U191>;
pub type U192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P192 = PInt<U192>; pub type N192 = NInt<U192>;
pub type U193 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P193 = PInt<U193>; pub type N193 = NInt<U193>;
pub type U194 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P194 = PInt<U194>; pub type N194 = NInt<U194>;
pub type U195 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P195 = PInt<U195>; pub type N195 = NInt<U195>;
pub type U196 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P196 = PInt<U196>; pub type N196 = NInt<U196>;
```

```
pub type U197 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P197 = PInt<U197>; pub type N197 = NInt<U197>;
pub type U198 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P198 = PInt<U198>; pub type N198 = NInt<U198>;
pub type U199 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P199 = PInt<U199>; pub type N199 = NInt<U199>;
pub type U200 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P200 = PInt<U200>; pub type N200 = NInt<U200>;
pub type U201 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P201 = PInt<U201>; pub type N201 = NInt<U201>;
pub type U202 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P202 = PInt<U202>; pub type N202 = NInt<U202>;
pub type U203 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P203 = PInt<U203>; pub type N203 = NInt<U203>;
pub type U204 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P204 = PInt<U204>; pub type N204 = NInt<U204>;
pub type U205 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P205 = PInt<U205>; pub type N205 = NInt<U205>;
pub type U206 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P206 = PInt<U206>; pub type N206 = NInt<U206>;
pub type U207 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P207 = PInt<U207>; pub type N207 = NInt<U207>;
pub type U208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P208 = PInt<U208>; pub type N208 = NInt<U208>;
pub type U209 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P209 = PInt<U209>; pub type N209 = NInt<U209>;
pub type U210 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P210 = PInt<U210>; pub type N210 = NInt<U210>;
pub type U211 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P211 = PInt<U211>; pub type N211 = NInt<U211>;
pub type U212 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P212 = PInt<U212>; pub type N212 = NInt<U212>;
pub type U213 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P213 = PInt<U213>; pub type N213 = NInt<U213>;
pub type U214 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P214 = PInt<U214>; pub type N214 = NInt<U214>;
pub type U215 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P215 = PInt<U215>; pub type N215 = NInt<U215>;
pub type U216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P216 = PInt<U216>; pub type N216 = NInt<U216>;
pub type U217 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P217 = PInt<U217>; pub type N217 = NInt<U217>;
pub type U218 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P218 = PInt<U218>; pub type N218 = NInt<U218>;
pub type U219 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P219 = PInt<U219>; pub type N219 = NInt<U219>;
pub type U220 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P220 = PInt<U220>; pub type N220 = NInt<U220>;
pub type U221 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P221 = PInt<U221>; pub type N221 = NInt<U221>;
pub type U222 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P222 = PInt<U222>; pub type N222 = NInt<U222>;
```

```
pub type U223 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P223 = PInt<U223>; pub type N223 = NInt<U223>;
pub type U224 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P224 = PInt<U224>; pub type N224 = NInt<U224>;
pub type U225 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P225 = PInt<U225>; pub type N225 = NInt<U225>;
pub type U226 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P226 = PInt<U226>; pub type N226 = NInt<U226>;
pub type U227 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P227 = PInt<U227>; pub type N227 = NInt<U227>;
pub type U228 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P228 = PInt<U228>; pub type N228 = NInt<U228>;
pub type U229 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P229 = PInt<U229>; pub type N229 = NInt<U229>;
pub type U230 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P230 = PInt<U230>; pub type N230 = NInt<U230>;
pub type U231 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P231 = PInt<U231>; pub type N231 = NInt<U231>;
pub type U232 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P232 = PInt<U232>; pub type N232 = NInt<U232>;
pub type U233 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P233 = PInt<U233>; pub type N233 = NInt<U233>;
pub type U234 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P234 = PInt<U234>; pub type N234 = NInt<U234>;
pub type U235 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P235 = PInt<U235>; pub type N235 = NInt<U235>;
pub type U236 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P236 = PInt<U236>; pub type N236 = NInt<U236>;
pub type U237 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P237 = PInt<U237>; pub type N237 = NInt<U237>;
pub type U238 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P238 = PInt<U238>; pub type N238 = NInt<U238>;
pub type U239 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P239 = PInt<U239>; pub type N239 = NInt<U239>;
pub type U240 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P240 = PInt<U240>; pub type N240 = NInt<U240>;
pub type U241 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P241 = PInt<U241>; pub type N241 = NInt<U241>;
pub type U242 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P242 = PInt<U242>; pub type N242 = NInt<U242>;
pub type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P243 = PInt<U243>; pub type N243 = NInt<U243>;
pub type U244 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P244 = PInt<U244>; pub type N244 = NInt<U244>;
pub type U245 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P245 = PInt<U245>; pub type N245 = NInt<U245>;
pub type U246 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P246 = PInt<U246>; pub type N246 = NInt<U246>;
pub type U247 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P247 = PInt<U247>; pub type N247 = NInt<U247>;
pub type U248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P248 = PInt<U248>; pub type N248 = NInt<U248>;
```

```
pub type U249 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P249 = PInt<U249>; pub type N249 = NInt<U249>;
pub type U250 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P250 = PInt<U250>; pub type N250 = NInt<U250>;
pub type U251 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P251 = PInt<U251>; pub type N251 = NInt<U251>;
pub type U252 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P252 = PInt<U252>; pub type N252 = NInt<U252>;
pub type U253 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P253 = PInt<U253>; pub type N253 = NInt<U253>;
pub type U254 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P254 = PInt<U254>; pub type N254 = NInt<U254>;
pub type U255 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P255 = PInt<U255>; pub type N255 = NInt<U255>;
pub type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P256 = PInt<U256>; pub type N256 = NInt<U256>;
pub type U257 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P257 = PInt<U257>; pub type N257 = NInt<U257>;
pub type U258 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P258 = PInt<U258>; pub type N258 = NInt<U258>;
pub type U259 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P259 = PInt<U259>; pub type N259 = NInt<U259>;
pub type U260 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P260 = PInt<U260>; pub type N260 = NInt<U260>;
pub type U261 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P261 = PInt<U261>; pub type N261 = NInt<U261>;
pub type U262 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P262 = PInt<U262>; pub type N262 = NInt<U262>;
pub type U263 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P263 = PInt<U263>; pub type N263 = NInt<U263>;
pub type U264 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P264 = PInt<U264>; pub type N264 = NInt<U264>;
pub type U265 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P265 = PInt<U265>; pub type N265 = NInt<U265>;
pub type U266 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P266 = PInt<U266>; pub type N266 = NInt<U266>;
pub type U267 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P267 = PInt<U267>; pub type N267 = NInt<U267>;
pub type U268 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P268 = PInt<U268>; pub type N268 = NInt<U268>;
pub type U269 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P269 = PInt<U269>; pub type N269 = NInt<U269>;
pub type U270 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P270 = PInt<U270>; pub type N270 = NInt<U270>;
pub type U271 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P271 = PInt<U271>; pub type N271 = NInt<U271>;
pub type U272 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P272 = PInt<U272>; pub type N272 = NInt<U272>;
pub type U273 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P273 = PInt<U273>; pub type N273 = NInt<U273>;
pub type U274 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P274 = PInt<U274>; pub type N274 = NInt<U274>;
```

```
pub type U275 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P275 = PInt<U275>; pub type N275 = NInt<U275>;
pub type U276 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P276 = PInt<U276>; pub type N276 = NInt<U276>;
pub type U277 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P277 = PInt<U277>; pub type N277 = NInt<U277>;
pub type U278 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P278 = PInt<U278>; pub type N278 = NInt<U278>;
pub type U279 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P279 = PInt<U279>; pub type N279 = NInt<U279>;
pub type U280 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P280 = PInt<U280>; pub type N280 = NInt<U280>;
pub type U281 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P281 = PInt<U281>; pub type N281 = NInt<U281>;
pub type U282 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P282 = PInt<U282>; pub type N282 = NInt<U282>;
pub type U283 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P283 = PInt<U283>; pub type N283 = NInt<U283>;
pub type U284 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P284 = PInt<U284>; pub type N284 = NInt<U284>;
pub type U285 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P285 = PInt<U285>; pub type N285 = NInt<U285>;
pub type U286 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P286 = PInt<U286>; pub type N286 = NInt<U286>;
pub type U287 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P287 = PInt<U287>; pub type N287 = NInt<U287>;
pub type U288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P288 = PInt<U288>; pub type N288 = NInt<U288>;
pub type U289 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P289 = PInt<U289>; pub type N289 = NInt<U289>;
pub type U290 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P290 = PInt<U290>; pub type N290 = NInt<U290>;
pub type U291 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P291 = PInt<U291>; pub type N291 = NInt<U291>;
pub type U292 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P292 = PInt<U292>; pub type N292 = NInt<U292>;
pub type U293 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P293 = PInt<U293>; pub type N293 = NInt<U293>;
pub type U294 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P294 = PInt<U294>; pub type N294 = NInt<U294>;
pub type U295 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P295 = PInt<U295>; pub type N295 = NInt<U295>;
pub type U296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P296 = PInt<U296>; pub type N296 = NInt<U296>;
pub type U297 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P297 = PInt<U297>; pub type N297 = NInt<U297>;
pub type U298 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P298 = PInt<U298>; pub type N298 = NInt<U298>;
pub type U299 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P299 = PInt<U299>; pub type N299 = NInt<U299>;
pub type U300 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P300 = PInt<U300>; pub type N300 = NInt<U300>;
```

```
pub type U301 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P301 = PInt<U301>; pub type N301 = NInt<U301>;
pub type U302 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P302 = PInt<U302>; pub type N302 = NInt<U302>;
pub type U303 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P303 = PInt<U303>; pub type N303 = NInt<U303>;
pub type U304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P304 = PInt<U304>; pub type N304 = NInt<U304>;
pub type U305 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P305 = PInt<U305>; pub type N305 = NInt<U305>;
pub type U306 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P306 = PInt<U306>; pub type N306 = NInt<U306>;
pub type U307 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P307 = PInt<U307>; pub type N307 = NInt<U307>;
pub type U308 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P308 = PInt<U308>; pub type N308 = NInt<U308>;
pub type U309 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P309 = PInt<U309>; pub type N309 = NInt<U309>;
pub type U310 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P310 = PInt<U310>; pub type N310 = NInt<U310>;
pub type U311 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P311 = PInt<U311>; pub type N311 = NInt<U311>;
pub type U312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P312 = PInt<U312>; pub type N312 = NInt<U312>;
pub type U313 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P313 = PInt<U313>; pub type N313 = NInt<U313>;
pub type U314 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P314 = PInt<U314>; pub type N314 = NInt<U314>;
pub type U315 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P315 = PInt<U315>; pub type N315 = NInt<U315>;
pub type U316 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P316 = PInt<U316>; pub type N316 = NInt<U316>;
pub type U317 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P317 = PInt<U317>; pub type N317 = NInt<U317>;
pub type U318 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P318 = PInt<U318>; pub type N318 = NInt<U318>;
pub type U319 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P319 = PInt<U319>; pub type N319 = NInt<U319>;
pub type U320 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P320 = PInt<U320>; pub type N320 = NInt<U320>;
pub type U321 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P321 = PInt<U321>; pub type N321 = NInt<U321>;
pub type U322 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P322 = PInt<U322>; pub type N322 = NInt<U322>;
pub type U323 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P323 = PInt<U323>; pub type N323 = NInt<U323>;
pub type U324 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P324 = PInt<U324>; pub type N324 = NInt<U324>;
pub type U325 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P325 = PInt<U325>; pub type N325 = NInt<U325>;
pub type U326 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P326 = PInt<U326>; pub type N326 = NInt<U326>;
```

```
pub type U327 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P327 = PInt<U327>; pub type N327 = NInt<U327>;
pub type U328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P328 = PInt<U328>; pub type N328 = NInt<U328>;
pub type U329 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P329 = PInt<U329>; pub type N329 = NInt<U329>;
pub type U330 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P330 = PInt<U330>; pub type N330 = NInt<U330>;
pub type U331 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P331 = PInt<U331>; pub type N331 = NInt<U331>;
pub type U332 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P332 = PInt<U332>; pub type N332 = NInt<U332>;
pub type U333 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P333 = PInt<U333>; pub type N333 = NInt<U333>;
pub type U334 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P334 = PInt<U334>; pub type N334 = NInt<U334>;
pub type U335 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P335 = PInt<U335>; pub type N335 = NInt<U335>;
pub type U336 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P336 = PInt<U336>; pub type N336 = NInt<U336>;
pub type U337 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P337 = PInt<U337>; pub type N337 = NInt<U337>;
pub type U338 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P338 = PInt<U338>; pub type N338 = NInt<U338>;
pub type U339 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P339 = PInt<U339>; pub type N339 = NInt<U339>;
pub type U340 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P340 = PInt<U340>; pub type N340 = NInt<U340>;
pub type U341 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P341 = PInt<U341>; pub type N341 = NInt<U341>;
pub type U342 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P342 = PInt<U342>; pub type N342 = NInt<U342>;
pub type U343 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P343 = PInt<U343>; pub type N343 = NInt<U343>;
pub type U344 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P344 = PInt<U344>; pub type N344 = NInt<U344>;
pub type U345 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P345 = PInt<U345>; pub type N345 = NInt<U345>;
pub type U346 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P346 = PInt<U346>; pub type N346 = NInt<U346>;
pub type U347 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P347 = PInt<U347>; pub type N347 = NInt<U347>;
pub type U348 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P348 = PInt<U348>; pub type N348 = NInt<U348>;
pub type U349 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P349 = PInt<U349>; pub type N349 = NInt<U349>;
pub type U350 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P350 = PInt<U350>; pub type N350 = NInt<U350>;
pub type U351 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P351 = PInt<U351>; pub type N351 = NInt<U351>;
pub type U352 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P352 = PInt<U352>; pub type N352 = NInt<U352>;
```

```
pub type U353 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P353 = PInt<U353>; pub type N353 = NInt<U353>;
pub type U354 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P354 = PInt<U354>; pub type N354 = NInt<U354>;
pub type U355 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P355 = PInt<U355>; pub type N355 = NInt<U355>;
pub type U356 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P356 = PInt<U356>; pub type N356 = NInt<U356>;
pub type U357 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P357 = PInt<U357>; pub type N357 = NInt<U357>;
pub type U358 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P358 = PInt<U358>; pub type N358 = NInt<U358>;
pub type U359 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P359 = PInt<U359>; pub type N359 = NInt<U359>;
pub type U360 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P360 = PInt<U360>; pub type N360 = NInt<U360>;
pub type U361 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P361 = PInt<U361>; pub type N361 = NInt<U361>;
pub type U362 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P362 = PInt<U362>; pub type N362 = NInt<U362>;
pub type U363 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P363 = PInt<U363>; pub type N363 = NInt<U363>;
pub type U364 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P364 = PInt<U364>; pub type N364 = NInt<U364>;
pub type U365 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P365 = PInt<U365>; pub type N365 = NInt<U365>;
pub type U366 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P366 = PInt<U366>; pub type N366 = NInt<U366>;
pub type U367 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P367 = PInt<U367>; pub type N367 = NInt<U367>;
pub type U368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P368 = PInt<U368>; pub type N368 = NInt<U368>;
pub type U369 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P369 = PInt<U369>; pub type N369 = NInt<U369>;
pub type U370 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P370 = PInt<U370>; pub type N370 = NInt<U370>;
pub type U371 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P371 = PInt<U371>; pub type N371 = NInt<U371>;
pub type U372 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P372 = PInt<U372>; pub type N372 = NInt<U372>;
pub type U373 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P373 = PInt<U373>; pub type N373 = NInt<U373>;
pub type U374 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P374 = PInt<U374>; pub type N374 = NInt<U374>;
pub type U375 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P375 = PInt<U375>; pub type N375 = NInt<U375>;
pub type U376 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P376 = PInt<U376>; pub type N376 = NInt<U376>;
pub type U377 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P377 = PInt<U377>; pub type N377 = NInt<U377>;
pub type U378 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P378 = PInt<U378>; pub type N378 = NInt<U378>;
```

```
pub type U379 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P379 = PInt<U379>; pub type N379 = NInt<U379>;
pub type U380 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P380 = PInt<U380>; pub type N380 = NInt<U380>;
pub type U381 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P381 = PInt<U381>; pub type N381 = NInt<U381>;
pub type U382 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P382 = PInt<U382>; pub type N382 = NInt<U382>;
pub type U383 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P383 = PInt<U383>; pub type N383 = NInt<U383>;
pub type U384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P384 = PInt<U384>; pub type N384 = NInt<U384>;
pub type U385 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P385 = PInt<U385>; pub type N385 = NInt<U385>;
pub type U386 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P386 = PInt<U386>; pub type N386 = NInt<U386>;
pub type U387 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P387 = PInt<U387>; pub type N387 = NInt<U387>;
pub type U388 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P388 = PInt<U388>; pub type N388 = NInt<U388>;
pub type U389 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P389 = PInt<U389>; pub type N389 = NInt<U389>;
pub type U390 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P390 = PInt<U390>; pub type N390 = NInt<U390>;
pub type U391 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P391 = PInt<U391>; pub type N391 = NInt<U391>;
pub type U392 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P392 = PInt<U392>; pub type N392 = NInt<U392>;
pub type U393 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P393 = PInt<U393>; pub type N393 = NInt<U393>;
pub type U394 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P394 = PInt<U394>; pub type N394 = NInt<U394>;
pub type U395 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P395 = PInt<U395>; pub type N395 = NInt<U395>;
pub type U396 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P396 = PInt<U396>; pub type N396 = NInt<U396>;
pub type U397 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P397 = PInt<U397>; pub type N397 = NInt<U397>;
pub type U398 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P398 = PInt<U398>; pub type N398 = NInt<U398>;
pub type U399 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P399 = PInt<U399>; pub type N399 = NInt<U399>;
pub type U400 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P400 = PInt<U400>; pub type N400 = NInt<U400>;
pub type U401 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P401 = PInt<U401>; pub type N401 = NInt<U401>;
pub type U402 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P402 = PInt<U402>; pub type N402 = NInt<U402>;
pub type U403 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P403 = PInt<U403>; pub type N403 = NInt<U403>;
pub type U404 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P404 = PInt<U404>; pub type N404 = NInt<U404>;
```

```
pub type U405 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P405 = PInt<U405>; pub type N405 = NInt<U405>;
pub type U406 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P406 = PInt<U406>; pub type N406 = NInt<U406>;
pub type U407 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P407 = PInt<U407>; pub type N407 = NInt<U407>;
pub type U408 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P408 = PInt<U408>; pub type N408 = NInt<U408>;
pub type U409 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P409 = PInt<U409>; pub type N409 = NInt<U409>;
pub type U410 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P410 = PInt<U410>; pub type N410 = NInt<U410>;
pub type U411 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P411 = PInt<U411>; pub type N411 = NInt<U411>;
pub type U412 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P412 = PInt<U412>; pub type N412 = NInt<U412>;
pub type U413 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P413 = PInt<U413>; pub type N413 = NInt<U413>;
pub type U414 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P414 = PInt<U414>; pub type N414 = NInt<U414>;
pub type U415 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P415 = PInt<U415>; pub type N415 = NInt<U415>;
pub type U416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P416 = PInt<U416>; pub type N416 = NInt<U416>;
pub type U417 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P417 = PInt<U417>; pub type N417 = NInt<U417>;
pub type U418 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P418 = PInt<U418>; pub type N418 = NInt<U418>;
pub type U419 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P419 = PInt<U419>; pub type N419 = NInt<U419>;
pub type U420 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P420 = PInt<U420>; pub type N420 = NInt<U420>;
pub type U421 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P421 = PInt<U421>; pub type N421 = NInt<U421>;
pub type U422 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P422 = PInt<U422>; pub type N422 = NInt<U422>;
pub type U423 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P423 = PInt<U423>; pub type N423 = NInt<U423>;
pub type U424 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P424 = PInt<U424>; pub type N424 = NInt<U424>;
pub type U425 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P425 = PInt<U425>; pub type N425 = NInt<U425>;
pub type U426 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P426 = PInt<U426>; pub type N426 = NInt<U426>;
pub type U427 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P427 = PInt<U427>; pub type N427 = NInt<U427>;
pub type U428 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P428 = PInt<U428>; pub type N428 = NInt<U428>;
pub type U429 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P429 = PInt<U429>; pub type N429 = NInt<U429>;
pub type U430 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P430 = PInt<U430>; pub type N430 = NInt<U430>;
```

```
pub type U431 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P431 = PInt<U431>; pub type N431 = NInt<U431>;
pub type U432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P432 = PInt<U432>; pub type N432 = NInt<U432>;
pub type U433 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P433 = PInt<U433>; pub type N433 = NInt<U433>;
pub type U434 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P434 = PInt<U434>; pub type N434 = NInt<U434>;
pub type U435 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P435 = PInt<U435>; pub type N435 = NInt<U435>;
pub type U436 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P436 = PInt<U436>; pub type N436 = NInt<U436>;
pub type U437 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P437 = PInt<U437>; pub type N437 = NInt<U437>;
pub type U438 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P438 = PInt<U438>; pub type N438 = NInt<U438>;
pub type U439 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P439 = PInt<U439>; pub type N439 = NInt<U439>;
pub type U440 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P440 = PInt<U440>; pub type N440 = NInt<U440>;
pub type U441 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P441 = PInt<U441>; pub type N441 = NInt<U441>;
pub type U442 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P442 = PInt<U442>; pub type N442 = NInt<U442>;
pub type U443 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P443 = PInt<U443>; pub type N443 = NInt<U443>;
pub type U444 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P444 = PInt<U444>; pub type N444 = NInt<U444>;
pub type U445 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P445 = PInt<U445>; pub type N445 = NInt<U445>;
pub type U446 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P446 = PInt<U446>; pub type N446 = NInt<U446>;
pub type U447 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P447 = PInt<U447>; pub type N447 = NInt<U447>;
pub type U448 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P448 = PInt<U448>; pub type N448 = NInt<U448>;
pub type U449 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P449 = PInt<U449>; pub type N449 = NInt<U449>;
pub type U450 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P450 = PInt<U450>; pub type N450 = NInt<U450>;
pub type U451 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P451 = PInt<U451>; pub type N451 = NInt<U451>;
pub type U452 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P452 = PInt<U452>; pub type N452 = NInt<U452>;
pub type U453 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P453 = PInt<U453>; pub type N453 = NInt<U453>;
pub type U454 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P454 = PInt<U454>; pub type N454 = NInt<U454>;
pub type U455 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P455 = PInt<U455>; pub type N455 = NInt<U455>;
pub type U456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P456 = PInt<U456>; pub type N456 = NInt<U456>;
```

```
pub type U457 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P457 = PInt<U457>; pub type N457 = NInt<U457>;
pub type U458 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P458 = PInt<U458>; pub type N458 = NInt<U458>;
pub type U459 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P459 = PInt<U459>; pub type N459 = NInt<U459>;
pub type U460 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P460 = PInt<U460>; pub type N460 = NInt<U460>;
pub type U461 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P461 = PInt<U461>; pub type N461 = NInt<U461>;
pub type U462 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P462 = PInt<U462>; pub type N462 = NInt<U462>;
pub type U463 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P463 = PInt<U463>; pub type N463 = NInt<U463>;
pub type U464 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P464 = PInt<U464>; pub type N464 = NInt<U464>;
pub type U465 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P465 = PInt<U465>; pub type N465 = NInt<U465>;
pub type U466 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P466 = PInt<U466>; pub type N466 = NInt<U466>;
pub type U467 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P467 = PInt<U467>; pub type N467 = NInt<U467>;
pub type U468 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P468 = PInt<U468>; pub type N468 = NInt<U468>;
pub type U469 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P469 = PInt<U469>; pub type N469 = NInt<U469>;
pub type U470 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P470 = PInt<U470>; pub type N470 = NInt<U470>;
pub type U471 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P471 = PInt<U471>; pub type N471 = NInt<U471>;
pub type U472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P472 = PInt<U472>; pub type N472 = NInt<U472>;
pub type U473 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P473 = PInt<U473>; pub type N473 = NInt<U473>;
pub type U474 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P474 = PInt<U474>; pub type N474 = NInt<U474>;
pub type U475 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P475 = PInt<U475>; pub type N475 = NInt<U475>;
pub type U476 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P476 = PInt<U476>; pub type N476 = NInt<U476>;
pub type U477 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P477 = PInt<U477>; pub type N477 = NInt<U477>;
pub type U478 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P478 = PInt<U478>; pub type N478 = NInt<U478>;
pub type U479 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P479 = PInt<U479>; pub type N479 = NInt<U479>;
pub type U480 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P480 = PInt<U480>; pub type N480 = NInt<U480>;
pub type U481 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P481 = PInt<U481>; pub type N481 = NInt<U481>;
pub type U482 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P482 = PInt<U482>; pub type N482 = NInt<U482>;
```

```
pub type U483 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P483 = PInt<U483>; pub type N483 = NInt<U483>;
pub type U484 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P484 = PInt<U484>; pub type N484 = NInt<U484>;
pub type U485 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P485 = PInt<U485>; pub type N485 = NInt<U485>;
pub type U486 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P486 = PInt<U486>; pub type N486 = NInt<U486>;
pub type U487 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P487 = PInt<U487>; pub type N487 = NInt<U487>;
pub type U488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P488 = PInt<U488>; pub type N488 = NInt<U488>;
pub type U489 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P489 = PInt<U489>; pub type N489 = NInt<U489>;
pub type U490 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P490 = PInt<U490>; pub type N490 = NInt<U490>;
pub type U491 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P491 = PInt<U491>; pub type N491 = NInt<U491>;
pub type U492 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P492 = PInt<U492>; pub type N492 = NInt<U492>;
pub type U493 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P493 = PInt<U493>; pub type N493 = NInt<U493>;
pub type U494 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P494 = PInt<U494>; pub type N494 = NInt<U494>;
pub type U495 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P495 = PInt<U495>; pub type N495 = NInt<U495>;
pub type U496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P496 = PInt<U496>; pub type N496 = NInt<U496>;
pub type U497 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P497 = PInt<U497>; pub type N497 = NInt<U497>;
pub type U498 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P498 = PInt<U498>; pub type N498 = NInt<U498>;
pub type U499 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P499 = PInt<U499>; pub type N499 = NInt<U499>;
pub type U500 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P500 = PInt<U500>; pub type N500 = NInt<U500>;
pub type U501 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P501 = PInt<U501>; pub type N501 = NInt<U501>;
pub type U502 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P502 = PInt<U502>; pub type N502 = NInt<U502>;
pub type U503 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P503 = PInt<U503>; pub type N503 = NInt<U503>;
pub type U504 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P504 = PInt<U504>; pub type N504 = NInt<U504>;
pub type U505 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P505 = PInt<U505>; pub type N505 = NInt<U505>;
pub type U506 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P506 = PInt<U506>; pub type N506 = NInt<U506>;
pub type U507 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P507 = PInt<U507>; pub type N507 = NInt<U507>;
pub type U508 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P508 = PInt<U508>; pub type N508 = NInt<U508>;
```

```
pub type U509 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P509 = PInt<U509>; pub type N509 = NInt<U509>;
pub type U510 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P510 = PInt<U510>; pub type N510 = NInt<U510>;
pub type U511 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P511 = PInt<U511>; pub type N511 = NInt<U511>;
pub type U512 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P512 = PInt<U512>; pub type N512 = NInt<U512>;
pub type U513 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P513 = PInt<U513>; pub type N513 = NInt<U513>;
pub type U514 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P514 = PInt<U514>; pub type N514 = NInt<U514>;
pub type U515 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P515 = PInt<U515>; pub type N515 = NInt<U515>;
pub type U516 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P516 = PInt<U516>; pub type N516 = NInt<U516>;
pub type U517 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P517 = PInt<U517>; pub type N517 = NInt<U517>;
pub type U518 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P518 = PInt<U518>; pub type N518 = NInt<U518>;
pub type U519 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P519 = PInt<U519>; pub type N519 = NInt<U519>;
pub type U520 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P520 = PInt<U520>; pub type N520 = NInt<U520>;
pub type U521 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P521 = PInt<U521>; pub type N521 = NInt<U521>;
pub type U522 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P522 = PInt<U522>; pub type N522 = NInt<U522>;
pub type U523 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P523 = PInt<U523>; pub type N523 = NInt<U523>;
pub type U524 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P524 = PInt<U524>; pub type N524 = NInt<U524>;
pub type U525 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P525 = PInt<U525>; pub type N525 = NInt<U525>;
pub type U526 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P526 = PInt<U526>; pub type N526 = NInt<U526>;
pub type U527 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P527 = PInt<U527>; pub type N527 = NInt<U527>;
pub type U528 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P528 = PInt<U528>; pub type N528 = NInt<U528>;
pub type U529 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P529 = PInt<U529>; pub type N529 = NInt<U529>;
pub type U530 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P530 = PInt<U530>; pub type N530 = NInt<U530>;
pub type U531 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P531 = PInt<U531>; pub type N531 = NInt<U531>;
pub type U532 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P532 = PInt<U532>; pub type N532 = NInt<U532>;
pub type U533 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P533 = PInt<U533>; pub type N533 = NInt<U533>;
pub type U534 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P534 = PInt<U534>; pub type N534 = NInt<U534>;
```

```
pub type U535 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P535 = PInt<U535>; pub type N535 = NInt<U535>;
pub type U536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P536 = PInt<U536>; pub type N536 = NInt<U536>;
pub type U537 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P537 = PInt<U537>; pub type N537 = NInt<U537>;
pub type U538 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P538 = PInt<U538>; pub type N538 = NInt<U538>;
pub type U539 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P539 = PInt<U539>; pub type N539 = NInt<U539>;
pub type U540 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P540 = PInt<U540>; pub type N540 = NInt<U540>;
pub type U541 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P541 = PInt<U541>; pub type N541 = NInt<U541>;
pub type U542 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P542 = PInt<U542>; pub type N542 = NInt<U542>;
pub type U543 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P543 = PInt<U543>; pub type N543 = NInt<U543>;
pub type U544 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P544 = PInt<U544>; pub type N544 = NInt<U544>;
pub type U545 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P545 = PInt<U545>; pub type N545 = NInt<U545>;
pub type U546 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P546 = PInt<U546>; pub type N546 = NInt<U546>;
pub type U547 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P547 = PInt<U547>; pub type N547 = NInt<U547>;
pub type U548 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P548 = PInt<U548>; pub type N548 = NInt<U548>;
pub type U549 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P549 = PInt<U549>; pub type N549 = NInt<U549>;
pub type U550 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P550 = PInt<U550>; pub type N550 = NInt<U550>;
pub type U551 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P551 = PInt<U551>; pub type N551 = NInt<U551>;
pub type U552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P552 = PInt<U552>; pub type N552 = NInt<U552>;
pub type U553 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P553 = PInt<U553>; pub type N553 = NInt<U553>;
pub type U554 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P554 = PInt<U554>; pub type N554 = NInt<U554>;
pub type U555 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P555 = PInt<U555>; pub type N555 = NInt<U555>;
pub type U556 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P556 = PInt<U556>; pub type N556 = NInt<U556>;
pub type U557 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P557 = PInt<U557>; pub type N557 = NInt<U557>;
pub type U558 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P558 = PInt<U558>; pub type N558 = NInt<U558>;
pub type U559 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P559 = PInt<U559>; pub type N559 = NInt<U559>;
pub type U560 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P560 = PInt<U560>; pub type N560 = NInt<U560>;
```

```
pub type U561 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P561 = PInt<U561>; pub type N561 = NInt<U561>;
pub type U562 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P562 = PInt<U562>; pub type N562 = NInt<U562>;
pub type U563 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P563 = PInt<U563>; pub type N563 = NInt<U563>;
pub type U564 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P564 = PInt<U564>; pub type N564 = NInt<U564>;
pub type U565 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P565 = PInt<U565>; pub type N565 = NInt<U565>;
pub type U566 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P566 = PInt<U566>; pub type N566 = NInt<U566>;
pub type U567 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P567 = PInt<U567>; pub type N567 = NInt<U567>;
pub type U568 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P568 = PInt<U568>; pub type N568 = NInt<U568>;
pub type U569 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P569 = PInt<U569>; pub type N569 = NInt<U569>;
pub type U570 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P570 = PInt<U570>; pub type N570 = NInt<U570>;
pub type U571 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P571 = PInt<U571>; pub type N571 = NInt<U571>;
pub type U572 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P572 = PInt<U572>; pub type N572 = NInt<U572>;
pub type U573 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P573 = PInt<U573>; pub type N573 = NInt<U573>;
pub type U574 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P574 = PInt<U574>; pub type N574 = NInt<U574>;
pub type U575 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P575 = PInt<U575>; pub type N575 = NInt<U575>;
pub type U576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P576 = PInt<U576>; pub type N576 = NInt<U576>;
pub type U577 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P577 = PInt<U577>; pub type N577 = NInt<U577>;
pub type U578 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P578 = PInt<U578>; pub type N578 = NInt<U578>;
pub type U579 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P579 = PInt<U579>; pub type N579 = NInt<U579>;
pub type U580 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P580 = PInt<U580>; pub type N580 = NInt<U580>;
pub type U581 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P581 = PInt<U581>; pub type N581 = NInt<U581>;
pub type U582 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P582 = PInt<U582>; pub type N582 = NInt<U582>;
pub type U583 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P583 = PInt<U583>; pub type N583 = NInt<U583>;
pub type U584 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P584 = PInt<U584>; pub type N584 = NInt<U584>;
pub type U585 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P585 = PInt<U585>; pub type N585 = NInt<U585>;
pub type U586 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P586 = PInt<U586>; pub type N586 = NInt<U586>;
```

```
pub type U587 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P587 = PInt<U587>; pub type N587 = NInt<U587>;
pub type U588 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P588 = PInt<U588>; pub type N588 = NInt<U588>;
pub type U589 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P589 = PInt<U589>; pub type N589 = NInt<U589>;
pub type U590 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P590 = PInt<U590>; pub type N590 = NInt<U590>;
pub type U591 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P591 = PInt<U591>; pub type N591 = NInt<U591>;
pub type U592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P592 = PInt<U592>; pub type N592 = NInt<U592>;
pub type U593 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P593 = PInt<U593>; pub type N593 = NInt<U593>;
pub type U594 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P594 = PInt<U594>; pub type N594 = NInt<U594>;
pub type U595 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P595 = PInt<U595>; pub type N595 = NInt<U595>;
pub type U596 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P596 = PInt<U596>; pub type N596 = NInt<U596>;
pub type U597 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P597 = PInt<U597>; pub type N597 = NInt<U597>;
pub type U598 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P598 = PInt<U598>; pub type N598 = NInt<U598>;
pub type U599 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P599 = PInt<U599>; pub type N599 = NInt<U599>;
pub type U600 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P600 = PInt<U600>; pub type N600 = NInt<U600>;
pub type U601 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P601 = PInt<U601>; pub type N601 = NInt<U601>;
pub type U602 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P602 = PInt<U602>; pub type N602 = NInt<U602>;
pub type U603 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P603 = PInt<U603>; pub type N603 = NInt<U603>;
pub type U604 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P604 = PInt<U604>; pub type N604 = NInt<U604>;
pub type U605 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P605 = PInt<U605>; pub type N605 = NInt<U605>;
pub type U606 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P606 = PInt<U606>; pub type N606 = NInt<U606>;
pub type U607 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P607 = PInt<U607>; pub type N607 = NInt<U607>;
pub type U608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P608 = PInt<U608>; pub type N608 = NInt<U608>;
pub type U609 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P609 = PInt<U609>; pub type N609 = NInt<U609>;
pub type U610 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P610 = PInt<U610>; pub type N610 = NInt<U610>;
pub type U611 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P611 = PInt<U611>; pub type N611 = NInt<U611>;
pub type U612 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P612 = PInt<U612>; pub type N612 = NInt<U612>;
```

```
pub type U613 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P613 = PInt<U613>; pub type N613 = NInt<U613>;
pub type U614 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P614 = PInt<U614>; pub type N614 = NInt<U614>;
pub type U615 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P615 = PInt<U615>; pub type N615 = NInt<U615>;
pub type U616 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P616 = PInt<U616>; pub type N616 = NInt<U616>;
pub type U617 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P617 = PInt<U617>; pub type N617 = NInt<U617>;
pub type U618 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P618 = PInt<U618>; pub type N618 = NInt<U618>;
pub type U619 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P619 = PInt<U619>; pub type N619 = NInt<U619>;
pub type U620 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P620 = PInt<U620>; pub type N620 = NInt<U620>;
pub type U621 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P621 = PInt<U621>; pub type N621 = NInt<U621>;
pub type U622 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P622 = PInt<U622>; pub type N622 = NInt<U622>;
pub type U623 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P623 = PInt<U623>; pub type N623 = NInt<U623>;
pub type U624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P624 = PInt<U624>; pub type N624 = NInt<U624>;
pub type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P625 = PInt<U625>; pub type N625 = NInt<U625>;
pub type U626 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P626 = PInt<U626>; pub type N626 = NInt<U626>;
pub type U627 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P627 = PInt<U627>; pub type N627 = NInt<U627>;
pub type U628 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P628 = PInt<U628>; pub type N628 = NInt<U628>;
pub type U629 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P629 = PInt<U629>; pub type N629 = NInt<U629>;
pub type U630 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P630 = PInt<U630>; pub type N630 = NInt<U630>;
pub type U631 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P631 = PInt<U631>; pub type N631 = NInt<U631>;
pub type U632 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P632 = PInt<U632>; pub type N632 = NInt<U632>;
pub type U633 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P633 = PInt<U633>; pub type N633 = NInt<U633>;
pub type U634 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P634 = PInt<U634>; pub type N634 = NInt<U634>;
pub type U635 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P635 = PInt<U635>; pub type N635 = NInt<U635>;
pub type U636 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P636 = PInt<U636>; pub type N636 = NInt<U636>;
pub type U637 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P637 = PInt<U637>; pub type N637 = NInt<U637>;
pub type U638 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P638 = PInt<U638>; pub type N638 = NInt<U638>;
```

```
pub type U639 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P639 = PInt<U639>; pub type N639 = NInt<U639>;
pub type U640 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P640 = PInt<U640>; pub type N640 = NInt<U640>;
pub type U641 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P641 = PInt<U641>; pub type N641 = NInt<U641>;
pub type U642 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P642 = PInt<U642>; pub type N642 = NInt<U642>;
pub type U643 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P643 = PInt<U643>; pub type N643 = NInt<U643>;
pub type U644 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P644 = PInt<U644>; pub type N644 = NInt<U644>;
pub type U645 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P645 = PInt<U645>; pub type N645 = NInt<U645>;
pub type U646 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P646 = PInt<U646>; pub type N646 = NInt<U646>;
pub type U647 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P647 = PInt<U647>; pub type N647 = NInt<U647>;
pub type U648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P648 = PInt<U648>; pub type N648 = NInt<U648>;
pub type U649 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P649 = PInt<U649>; pub type N649 = NInt<U649>;
pub type U650 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P650 = PInt<U650>; pub type N650 = NInt<U650>;
pub type U651 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P651 = PInt<U651>; pub type N651 = NInt<U651>;
pub type U652 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P652 = PInt<U652>; pub type N652 = NInt<U652>;
pub type U653 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P653 = PInt<U653>; pub type N653 = NInt<U653>;
pub type U654 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P654 = PInt<U654>; pub type N654 = NInt<U654>;
pub type U655 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P655 = PInt<U655>; pub type N655 = NInt<U655>;
pub type U656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P656 = PInt<U656>; pub type N656 = NInt<U656>;
pub type U657 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P657 = PInt<U657>; pub type N657 = NInt<U657>;
pub type U658 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P658 = PInt<U658>; pub type N658 = NInt<U658>;
pub type U659 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P659 = PInt<U659>; pub type N659 = NInt<U659>;
pub type U660 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P660 = PInt<U660>; pub type N660 = NInt<U660>;
pub type U661 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P661 = PInt<U661>; pub type N661 = NInt<U661>;
pub type U662 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P662 = PInt<U662>; pub type N662 = NInt<U662>;
pub type U663 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P663 = PInt<U663>; pub type N663 = NInt<U663>;
pub type U664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P664 = PInt<U664>; pub type N664 = NInt<U664>;
```

```
pub type U665 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P665 = PInt<U665>; pub type N665 = NInt<U665>;
pub type U666 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P666 = PInt<U666>; pub type N666 = NInt<U666>;
pub type U667 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P667 = PInt<U667>; pub type N667 = NInt<U667>;
pub type U668 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P668 = PInt<U668>; pub type N668 = NInt<U668>;
pub type U669 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P669 = PInt<U669>; pub type N669 = NInt<U669>;
pub type U670 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P670 = PInt<U670>; pub type N670 = NInt<U670>;
pub type U671 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P671 = PInt<U671>; pub type N671 = NInt<U671>;
pub type U672 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P672 = PInt<U672>; pub type N672 = NInt<U672>;
pub type U673 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P673 = PInt<U673>; pub type N673 = NInt<U673>;
pub type U674 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P674 = PInt<U674>; pub type N674 = NInt<U674>;
pub type U675 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P675 = PInt<U675>; pub type N675 = NInt<U675>;
pub type U676 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P676 = PInt<U676>; pub type N676 = NInt<U676>;
pub type U677 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P677 = PInt<U677>; pub type N677 = NInt<U677>;
pub type U678 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P678 = PInt<U678>; pub type N678 = NInt<U678>;
pub type U679 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P679 = PInt<U679>; pub type N679 = NInt<U679>;
pub type U680 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P680 = PInt<U680>; pub type N680 = NInt<U680>;
pub type U681 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P681 = PInt<U681>; pub type N681 = NInt<U681>;
pub type U682 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P682 = PInt<U682>; pub type N682 = NInt<U682>;
pub type U683 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P683 = PInt<U683>; pub type N683 = NInt<U683>;
pub type U684 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P684 = PInt<U684>; pub type N684 = NInt<U684>;
pub type U685 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P685 = PInt<U685>; pub type N685 = NInt<U685>;
pub type U686 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P686 = PInt<U686>; pub type N686 = NInt<U686>;
pub type U687 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P687 = PInt<U687>; pub type N687 = NInt<U687>;
pub type U688 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P688 = PInt<U688>; pub type N688 = NInt<U688>;
pub type U689 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P689 = PInt<U689>; pub type N689 = NInt<U689>;
pub type U690 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P690 = PInt<U690>; pub type N690 = NInt<U690>;
```

```
pub type U691 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P691 = PInt<U691>; pub type N691 = NInt<U691>;
pub type U692 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P692 = PInt<U692>; pub type N692 = NInt<U692>;
pub type U693 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P693 = PInt<U693>; pub type N693 = NInt<U693>;
pub type U694 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P694 = PInt<U694>; pub type N694 = NInt<U694>;
pub type U695 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P695 = PInt<U695>; pub type N695 = NInt<U695>;
pub type U696 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P696 = PInt<U696>; pub type N696 = NInt<U696>;
pub type U697 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P697 = PInt<U697>; pub type N697 = NInt<U697>;
pub type U698 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P698 = PInt<U698>; pub type N698 = NInt<U698>;
pub type U699 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P699 = PInt<U699>; pub type N699 = NInt<U699>;
pub type U700 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P700 = PInt<U700>; pub type N700 = NInt<U700>;
pub type U701 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P701 = PInt<U701>; pub type N701 = NInt<U701>;
pub type U702 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P702 = PInt<U702>; pub type N702 = NInt<U702>;
pub type U703 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P703 = PInt<U703>; pub type N703 = NInt<U703>;
pub type U704 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P704 = PInt<U704>; pub type N704 = NInt<U704>;
pub type U705 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P705 = PInt<U705>; pub type N705 = NInt<U705>;
pub type U706 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P706 = PInt<U706>; pub type N706 = NInt<U706>;
pub type U707 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P707 = PInt<U707>; pub type N707 = NInt<U707>;
pub type U708 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P708 = PInt<U708>; pub type N708 = NInt<U708>;
pub type U709 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P709 = PInt<U709>; pub type N709 = NInt<U709>;
pub type U710 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P710 = PInt<U710>; pub type N710 = NInt<U710>;
pub type U711 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P711 = PInt<U711>; pub type N711 = NInt<U711>;
pub type U712 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P712 = PInt<U712>; pub type N712 = NInt<U712>;
pub type U713 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P713 = PInt<U713>; pub type N713 = NInt<U713>;
pub type U714 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P714 = PInt<U714>; pub type N714 = NInt<U714>;
pub type U715 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P715 = PInt<U715>; pub type N715 = NInt<U715>;
pub type U716 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P716 = PInt<U716>; pub type N716 = NInt<U716>;
```

```
pub type U717 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P717 = PInt<U717>; pub type N717 = NInt<U717>;
pub type U718 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P718 = PInt<U718>; pub type N718 = NInt<U718>;
pub type U719 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P719 = PInt<U719>; pub type N719 = NInt<U719>;
pub type U720 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P720 = PInt<U720>; pub type N720 = NInt<U720>;
pub type U721 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P721 = PInt<U721>; pub type N721 = NInt<U721>;
pub type U722 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P722 = PInt<U722>; pub type N722 = NInt<U722>;
pub type U723 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P723 = PInt<U723>; pub type N723 = NInt<U723>;
pub type U724 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P724 = PInt<U724>; pub type N724 = NInt<U724>;
pub type U725 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P725 = PInt<U725>; pub type N725 = NInt<U725>;
pub type U726 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P726 = PInt<U726>; pub type N726 = NInt<U726>;
pub type U727 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P727 = PInt<U727>; pub type N727 = NInt<U727>;
pub type U728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P728 = PInt<U728>; pub type N728 = NInt<U728>;
pub type U729 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P729 = PInt<U729>; pub type N729 = NInt<U729>;
pub type U730 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P730 = PInt<U730>; pub type N730 = NInt<U730>;
pub type U731 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P731 = PInt<U731>; pub type N731 = NInt<U731>;
pub type U732 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P732 = PInt<U732>; pub type N732 = NInt<U732>;
pub type U733 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P733 = PInt<U733>; pub type N733 = NInt<U733>;
pub type U734 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P734 = PInt<U734>; pub type N734 = NInt<U734>;
pub type U735 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P735 = PInt<U735>; pub type N735 = NInt<U735>;
pub type U736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P736 = PInt<U736>; pub type N736 = NInt<U736>;
pub type U737 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P737 = PInt<U737>; pub type N737 = NInt<U737>;
pub type U738 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P738 = PInt<U738>; pub type N738 = NInt<U738>;
pub type U739 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P739 = PInt<U739>; pub type N739 = NInt<U739>;
pub type U740 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P740 = PInt<U740>; pub type N740 = NInt<U740>;
pub type U741 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P741 = PInt<U741>; pub type N741 = NInt<U741>;
pub type U742 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P742 = PInt<U742>; pub type N742 = NInt<U742>;
```

```
pub type U743 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P743 = PInt<U743>; pub type N743 = NInt<U743>;
pub type U744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P744 = PInt<U744>; pub type N744 = NInt<U744>;
pub type U745 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P745 = PInt<U745>; pub type N745 = NInt<U745>;
pub type U746 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P746 = PInt<U746>; pub type N746 = NInt<U746>;
pub type U747 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P747 = PInt<U747>; pub type N747 = NInt<U747>;
pub type U748 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P748 = PInt<U748>; pub type N748 = NInt<U748>;
pub type U749 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P749 = PInt<U749>; pub type N749 = NInt<U749>;
pub type U750 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P750 = PInt<U750>; pub type N750 = NInt<U750>;
pub type U751 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P751 = PInt<U751>; pub type N751 = NInt<U751>;
pub type U752 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P752 = PInt<U752>; pub type N752 = NInt<U752>;
pub type U753 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P753 = PInt<U753>; pub type N753 = NInt<U753>;
pub type U754 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P754 = PInt<U754>; pub type N754 = NInt<U754>;
pub type U755 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P755 = PInt<U755>; pub type N755 = NInt<U755>;
pub type U756 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P756 = PInt<U756>; pub type N756 = NInt<U756>;
pub type U757 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P757 = PInt<U757>; pub type N757 = NInt<U757>;
pub type U758 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P758 = PInt<U758>; pub type N758 = NInt<U758>;
pub type U759 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P759 = PInt<U759>; pub type N759 = NInt<U759>;
pub type U760 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P760 = PInt<U760>; pub type N760 = NInt<U760>;
pub type U761 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P761 = PInt<U761>; pub type N761 = NInt<U761>;
pub type U762 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P762 = PInt<U762>; pub type N762 = NInt<U762>;
pub type U763 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P763 = PInt<U763>; pub type N763 = NInt<U763>;
pub type U764 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P764 = PInt<U764>; pub type N764 = NInt<U764>;
pub type U765 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P765 = PInt<U765>; pub type N765 = NInt<U765>;
pub type U766 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P766 = PInt<U766>; pub type N766 = NInt<U766>;
pub type U767 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P767 = PInt<U767>; pub type N767 = NInt<U767>;
pub type U768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P768 = PInt<U768>; pub type N768 = NInt<U768>;
```

```
pub type U769 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P769 = PInt<U769>; pub type N769 = NInt<U769>;
pub type U770 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P770 = PInt<U770>; pub type N770 = NInt<U770>;
pub type U771 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P771 = PInt<U771>; pub type N771 = NInt<U771>;
pub type U772 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P772 = PInt<U772>; pub type N772 = NInt<U772>;
pub type U773 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P773 = PInt<U773>; pub type N773 = NInt<U773>;
pub type U774 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P774 = PInt<U774>; pub type N774 = NInt<U774>;
pub type U775 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P775 = PInt<U775>; pub type N775 = NInt<U775>;
pub type U776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P776 = PInt<U776>; pub type N776 = NInt<U776>;
pub type U777 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P777 = PInt<U777>; pub type N777 = NInt<U777>;
pub type U778 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P778 = PInt<U778>; pub type N778 = NInt<U778>;
pub type U779 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P779 = PInt<U779>; pub type N779 = NInt<U779>;
pub type U780 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P780 = PInt<U780>; pub type N780 = NInt<U780>;
pub type U781 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P781 = PInt<U781>; pub type N781 = NInt<U781>;
pub type U782 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P782 = PInt<U782>; pub type N782 = NInt<U782>;
pub type U783 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P783 = PInt<U783>; pub type N783 = NInt<U783>;
pub type U784 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P784 = PInt<U784>; pub type N784 = NInt<U784>;
pub type U785 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P785 = PInt<U785>; pub type N785 = NInt<U785>;
pub type U786 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P786 = PInt<U786>; pub type N786 = NInt<U786>;
pub type U787 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P787 = PInt<U787>; pub type N787 = NInt<U787>;
pub type U788 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P788 = PInt<U788>; pub type N788 = NInt<U788>;
pub type U789 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P789 = PInt<U789>; pub type N789 = NInt<U789>;
pub type U790 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P790 = PInt<U790>; pub type N790 = NInt<U790>;
pub type U791 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P791 = PInt<U791>; pub type N791 = NInt<U791>;
pub type U792 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P792 = PInt<U792>; pub type N792 = NInt<U792>;
pub type U793 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P793 = PInt<U793>; pub type N793 = NInt<U793>;
pub type U794 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P794 = PInt<U794>; pub type N794 = NInt<U794>;
```

```
pub type U795 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P795 = PInt<U795>; pub type N795 = NInt<U795>;
pub type U796 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P796 = PInt<U796>; pub type N796 = NInt<U796>;
pub type U797 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P797 = PInt<U797>; pub type N797 = NInt<U797>;
pub type U798 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P798 = PInt<U798>; pub type N798 = NInt<U798>;
pub type U799 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P799 = PInt<U799>; pub type N799 = NInt<U799>;
pub type U800 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P800 = PInt<U800>; pub type N800 = NInt<U800>;
pub type U801 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P801 = PInt<U801>; pub type N801 = NInt<U801>;
pub type U802 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P802 = PInt<U802>; pub type N802 = NInt<U802>;
pub type U803 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P803 = PInt<U803>; pub type N803 = NInt<U803>;
pub type U804 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P804 = PInt<U804>; pub type N804 = NInt<U804>;
pub type U805 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P805 = PInt<U805>; pub type N805 = NInt<U805>;
pub type U806 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P806 = PInt<U806>; pub type N806 = NInt<U806>;
pub type U807 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P807 = PInt<U807>; pub type N807 = NInt<U807>;
pub type U808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P808 = PInt<U808>; pub type N808 = NInt<U808>;
pub type U809 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P809 = PInt<U809>; pub type N809 = NInt<U809>;
pub type U810 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P810 = PInt<U810>; pub type N810 = NInt<U810>;
pub type U811 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P811 = PInt<U811>; pub type N811 = NInt<U811>;
pub type U812 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P812 = PInt<U812>; pub type N812 = NInt<U812>;
pub type U813 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P813 = PInt<U813>; pub type N813 = NInt<U813>;
pub type U814 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P814 = PInt<U814>; pub type N814 = NInt<U814>;
pub type U815 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P815 = PInt<U815>; pub type N815 = NInt<U815>;
pub type U816 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P816 = PInt<U816>; pub type N816 = NInt<U816>;
pub type U817 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P817 = PInt<U817>; pub type N817 = NInt<U817>;
pub type U818 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P818 = PInt<U818>; pub type N818 = NInt<U818>;
pub type U819 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P819 = PInt<U819>; pub type N819 = NInt<U819>;
pub type U820 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P820 = PInt<U820>; pub type N820 = NInt<U820>;
```

```
pub type U821 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P821 = PInt<U821>; pub type N821 = NInt<U821>;
pub type U822 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P822 = PInt<U822>; pub type N822 = NInt<U822>;
pub type U823 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P823 = PInt<U823>; pub type N823 = NInt<U823>;
pub type U824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P824 = PInt<U824>; pub type N824 = NInt<U824>;
pub type U825 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P825 = PInt<U825>; pub type N825 = NInt<U825>;
pub type U826 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P826 = PInt<U826>; pub type N826 = NInt<U826>;
pub type U827 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P827 = PInt<U827>; pub type N827 = NInt<U827>;
pub type U828 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P828 = PInt<U828>; pub type N828 = NInt<U828>;
pub type U829 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P829 = PInt<U829>; pub type N829 = NInt<U829>;
pub type U830 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P830 = PInt<U830>; pub type N830 = NInt<U830>;
pub type U831 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P831 = PInt<U831>; pub type N831 = NInt<U831>;
pub type U832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P832 = PInt<U832>; pub type N832 = NInt<U832>;
pub type U833 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P833 = PInt<U833>; pub type N833 = NInt<U833>;
pub type U834 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P834 = PInt<U834>; pub type N834 = NInt<U834>;
pub type U835 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P835 = PInt<U835>; pub type N835 = NInt<U835>;
pub type U836 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P836 = PInt<U836>; pub type N836 = NInt<U836>;
pub type U837 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P837 = PInt<U837>; pub type N837 = NInt<U837>;
pub type U838 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P838 = PInt<U838>; pub type N838 = NInt<U838>;
pub type U839 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P839 = PInt<U839>; pub type N839 = NInt<U839>;
pub type U840 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P840 = PInt<U840>; pub type N840 = NInt<U840>;
pub type U841 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P841 = PInt<U841>; pub type N841 = NInt<U841>;
pub type U842 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P842 = PInt<U842>; pub type N842 = NInt<U842>;
pub type U843 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P843 = PInt<U843>; pub type N843 = NInt<U843>;
pub type U844 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P844 = PInt<U844>; pub type N844 = NInt<U844>;
pub type U845 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P845 = PInt<U845>; pub type N845 = NInt<U845>;
pub type U846 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P846 = PInt<U846>; pub type N846 = NInt<U846>;
```

```
pub type U847 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P847 = PInt<U847>; pub type N847 = NInt<U847>;
pub type U848 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P848 = PInt<U848>; pub type N848 = NInt<U848>;
pub type U849 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P849 = PInt<U849>; pub type N849 = NInt<U849>;
pub type U850 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P850 = PInt<U850>; pub type N850 = NInt<U850>;
pub type U851 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P851 = PInt<U851>; pub type N851 = NInt<U851>;
pub type U852 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P852 = PInt<U852>; pub type N852 = NInt<U852>;
pub type U853 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P853 = PInt<U853>; pub type N853 = NInt<U853>;
pub type U854 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P854 = PInt<U854>; pub type N854 = NInt<U854>;
pub type U855 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P855 = PInt<U855>; pub type N855 = NInt<U855>;
pub type U856 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P856 = PInt<U856>; pub type N856 = NInt<U856>;
pub type U857 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P857 = PInt<U857>; pub type N857 = NInt<U857>;
pub type U858 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P858 = PInt<U858>; pub type N858 = NInt<U858>;
pub type U859 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P859 = PInt<U859>; pub type N859 = NInt<U859>;
pub type U860 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P860 = PInt<U860>; pub type N860 = NInt<U860>;
pub type U861 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P861 = PInt<U861>; pub type N861 = NInt<U861>;
pub type U862 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P862 = PInt<U862>; pub type N862 = NInt<U862>;
pub type U863 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P863 = PInt<U863>; pub type N863 = NInt<U863>;
pub type U864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P864 = PInt<U864>; pub type N864 = NInt<U864>;
pub type U865 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P865 = PInt<U865>; pub type N865 = NInt<U865>;
pub type U866 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P866 = PInt<U866>; pub type N866 = NInt<U866>;
pub type U867 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P867 = PInt<U867>; pub type N867 = NInt<U867>;
pub type U868 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P868 = PInt<U868>; pub type N868 = NInt<U868>;
pub type U869 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P869 = PInt<U869>; pub type N869 = NInt<U869>;
pub type U870 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P870 = PInt<U870>; pub type N870 = NInt<U870>;
pub type U871 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P871 = PInt<U871>; pub type N871 = NInt<U871>;
pub type U872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P872 = PInt<U872>; pub type N872 = NInt<U872>;
```

```
pub type U873 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P873 = PInt<U873>; pub type N873 = NInt<U873>;
pub type U874 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P874 = PInt<U874>; pub type N874 = NInt<U874>;
pub type U875 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P875 = PInt<U875>; pub type N875 = NInt<U875>;
pub type U876 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P876 = PInt<U876>; pub type N876 = NInt<U876>;
pub type U877 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P877 = PInt<U877>; pub type N877 = NInt<U877>;
pub type U878 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P878 = PInt<U878>; pub type N878 = NInt<U878>;
pub type U879 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P879 = PInt<U879>; pub type N879 = NInt<U879>;
pub type U880 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P880 = PInt<U880>; pub type N880 = NInt<U880>;
pub type U881 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P881 = PInt<U881>; pub type N881 = NInt<U881>;
pub type U882 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P882 = PInt<U882>; pub type N882 = NInt<U882>;
pub type U883 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P883 = PInt<U883>; pub type N883 = NInt<U883>;
pub type U884 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P884 = PInt<U884>; pub type N884 = NInt<U884>;
pub type U885 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P885 = PInt<U885>; pub type N885 = NInt<U885>;
pub type U886 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P886 = PInt<U886>; pub type N886 = NInt<U886>;
pub type U887 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P887 = PInt<U887>; pub type N887 = NInt<U887>;
pub type U888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P888 = PInt<U888>; pub type N888 = NInt<U888>;
pub type U889 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P889 = PInt<U889>; pub type N889 = NInt<U889>;
pub type U890 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P890 = PInt<U890>; pub type N890 = NInt<U890>;
pub type U891 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P891 = PInt<U891>; pub type N891 = NInt<U891>;
pub type U892 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P892 = PInt<U892>; pub type N892 = NInt<U892>;
pub type U893 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P893 = PInt<U893>; pub type N893 = NInt<U893>;
pub type U894 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P894 = PInt<U894>; pub type N894 = NInt<U894>;
pub type U895 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P895 = PInt<U895>; pub type N895 = NInt<U895>;
pub type U896 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P896 = PInt<U896>; pub type N896 = NInt<U896>;
pub type U897 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P897 = PInt<U897>; pub type N897 = NInt<U897>;
pub type U898 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P898 = PInt<U898>; pub type N898 = NInt<U898>;
```

```
pub type U899 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P899 = PInt<U899>; pub type N899 = NInt<U899>;
pub type U900 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P900 = PInt<U900>; pub type N900 = NInt<U900>;
pub type U901 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P901 = PInt<U901>; pub type N901 = NInt<U901>;
pub type U902 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P902 = PInt<U902>; pub type N902 = NInt<U902>;
pub type U903 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P903 = PInt<U903>; pub type N903 = NInt<U903>;
pub type U904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P904 = PInt<U904>; pub type N904 = NInt<U904>;
pub type U905 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P905 = PInt<U905>; pub type N905 = NInt<U905>;
pub type U906 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P906 = PInt<U906>; pub type N906 = NInt<U906>;
pub type U907 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P907 = PInt<U907>; pub type N907 = NInt<U907>;
pub type U908 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P908 = PInt<U908>; pub type N908 = NInt<U908>;
pub type U909 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P909 = PInt<U909>; pub type N909 = NInt<U909>;
pub type U910 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P910 = PInt<U910>; pub type N910 = NInt<U910>;
pub type U911 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P911 = PInt<U911>; pub type N911 = NInt<U911>;
pub type U912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P912 = PInt<U912>; pub type N912 = NInt<U912>;
pub type U913 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P913 = PInt<U913>; pub type N913 = NInt<U913>;
pub type U914 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P914 = PInt<U914>; pub type N914 = NInt<U914>;
pub type U915 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P915 = PInt<U915>; pub type N915 = NInt<U915>;
pub type U916 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P916 = PInt<U916>; pub type N916 = NInt<U916>;
pub type U917 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P917 = PInt<U917>; pub type N917 = NInt<U917>;
pub type U918 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P918 = PInt<U918>; pub type N918 = NInt<U918>;
pub type U919 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P919 = PInt<U919>; pub type N919 = NInt<U919>;
pub type U920 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P920 = PInt<U920>; pub type N920 = NInt<U920>;
pub type U921 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P921 = PInt<U921>; pub type N921 = NInt<U921>;
pub type U922 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P922 = PInt<U922>; pub type N922 = NInt<U922>;
pub type U923 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P923 = PInt<U923>; pub type N923 = NInt<U923>;
pub type U924 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P924 = PInt<U924>; pub type N924 = NInt<U924>;
```

```
pub type U925 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P925 = PInt<U925>; pub type N925 = NInt<U925>;
pub type U926 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P926 = PInt<U926>; pub type N926 = NInt<U926>;
pub type U927 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P927 = PInt<U927>; pub type N927 = NInt<U927>;
pub type U928 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P928 = PInt<U928>; pub type N928 = NInt<U928>;
pub type U929 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P929 = PInt<U929>; pub type N929 = NInt<U929>;
pub type U930 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P930 = PInt<U930>; pub type N930 = NInt<U930>;
pub type U931 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P931 = PInt<U931>; pub type N931 = NInt<U931>;
pub type U932 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P932 = PInt<U932>; pub type N932 = NInt<U932>;
pub type U933 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P933 = PInt<U933>; pub type N933 = NInt<U933>;
pub type U934 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P934 = PInt<U934>; pub type N934 = NInt<U934>;
pub type U935 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P935 = PInt<U935>; pub type N935 = NInt<U935>;
pub type U936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P936 = PInt<U936>; pub type N936 = NInt<U936>;
pub type U937 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P937 = PInt<U937>; pub type N937 = NInt<U937>;
pub type U938 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P938 = PInt<U938>; pub type N938 = NInt<U938>;
pub type U939 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P939 = PInt<U939>; pub type N939 = NInt<U939>;
pub type U940 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P940 = PInt<U940>; pub type N940 = NInt<U940>;
pub type U941 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P941 = PInt<U941>; pub type N941 = NInt<U941>;
pub type U942 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P942 = PInt<U942>; pub type N942 = NInt<U942>;
pub type U943 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P943 = PInt<U943>; pub type N943 = NInt<U943>;
pub type U944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P944 = PInt<U944>; pub type N944 = NInt<U944>;
pub type U945 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P945 = PInt<U945>; pub type N945 = NInt<U945>;
pub type U946 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P946 = PInt<U946>; pub type N946 = NInt<U946>;
pub type U947 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P947 = PInt<U947>; pub type N947 = NInt<U947>;
pub type U948 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P948 = PInt<U948>; pub type N948 = NInt<U948>;
pub type U949 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P949 = PInt<U949>; pub type N949 = NInt<U949>;
pub type U950 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P950 = PInt<U950>; pub type N950 = NInt<U950>;
```

```
pub type U951 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P951 = PInt<U951>; pub type N951 = NInt<U951>;
pub type U952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P952 = PInt<U952>; pub type N952 = NInt<U952>;
pub type U953 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P953 = PInt<U953>; pub type N953 = NInt<U953>;
pub type U954 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P954 = PInt<U954>; pub type N954 = NInt<U954>;
pub type U955 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P955 = PInt<U955>; pub type N955 = NInt<U955>;
pub type U956 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P956 = PInt<U956>; pub type N956 = NInt<U956>;
pub type U957 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P957 = PInt<U957>; pub type N957 = NInt<U957>;
pub type U958 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P958 = PInt<U958>; pub type N958 = NInt<U958>;
pub type U959 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P959 = PInt<U959>; pub type N959 = NInt<U959>;
pub type U960 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P960 = PInt<U960>; pub type N960 = NInt<U960>;
pub type U961 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P961 = PInt<U961>; pub type N961 = NInt<U961>;
pub type U962 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P962 = PInt<U962>; pub type N962 = NInt<U962>;
pub type U963 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P963 = PInt<U963>; pub type N963 = NInt<U963>;
pub type U964 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P964 = PInt<U964>; pub type N964 = NInt<U964>;
pub type U965 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P965 = PInt<U965>; pub type N965 = NInt<U965>;
pub type U966 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P966 = PInt<U966>; pub type N966 = NInt<U966>;
pub type U967 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P967 = PInt<U967>; pub type N967 = NInt<U967>;
pub type U968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P968 = PInt<U968>; pub type N968 = NInt<U968>;
pub type U969 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P969 = PInt<U969>; pub type N969 = NInt<U969>;
pub type U970 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P970 = PInt<U970>; pub type N970 = NInt<U970>;
pub type U971 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P971 = PInt<U971>; pub type N971 = NInt<U971>;
pub type U972 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P972 = PInt<U972>; pub type N972 = NInt<U972>;
pub type U973 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P973 = PInt<U973>; pub type N973 = NInt<U973>;
pub type U974 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P974 = PInt<U974>; pub type N974 = NInt<U974>;
pub type U975 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P975 = PInt<U975>; pub type N975 = NInt<U975>;
pub type U976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P976 = PInt<U976>; pub type N976 = NInt<U976>;
```

```
pub type U977 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P977 = PInt<U977>; pub type N977 = NInt<U977>;
pub type U978 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P978 = PInt<U978>; pub type N978 = NInt<U978>;
pub type U979 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P979 = PInt<U979>; pub type N979 = NInt<U979>;
pub type U980 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P980 = PInt<U980>; pub type N980 = NInt<U980>;
pub type U981 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P981 = PInt<U981>; pub type N981 = NInt<U981>;
pub type U982 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P982 = PInt<U982>; pub type N982 = NInt<U982>;
pub type U983 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P983 = PInt<U983>; pub type N983 = NInt<U983>;
pub type U984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P984 = PInt<U984>; pub type N984 = NInt<U984>;
pub type U985 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P985 = PInt<U985>; pub type N985 = NInt<U985>;
pub type U986 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P986 = PInt<U986>; pub type N986 = NInt<U986>;
pub type U987 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P987 = PInt<U987>; pub type N987 = NInt<U987>;
pub type U988 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P988 = PInt<U988>; pub type N988 = NInt<U988>;
pub type U989 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P989 = PInt<U989>; pub type N989 = NInt<U989>;
pub type U990 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P990 = PInt<U990>; pub type N990 = NInt<U990>;
pub type U991 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P991 = PInt<U991>; pub type N991 = NInt<U991>;
pub type U992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P992 = PInt<U992>; pub type N992 = NInt<U992>;
pub type U993 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P993 = PInt<U993>; pub type N993 = NInt<U993>;
pub type U994 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P994 = PInt<U994>; pub type N994 = NInt<U994>;
pub type U995 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P995 = PInt<U995>; pub type N995 = NInt<U995>;
pub type U996 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P996 = PInt<U996>; pub type N996 = NInt<U996>;
pub type U997 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P997 = PInt<U997>; pub type N997 = NInt<U997>;
pub type U998 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P998 = PInt<U998>; pub type N998 = NInt<U998>;
pub type U999 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P999 = PInt<U999>; pub type N999 = NInt<U999>;
pub type U1000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1000 = PInt<U1000>; pub type N1000 = NInt<U1000>;
pub type U1001 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1001 = PInt<U1001>; pub type N1001 = NInt<U1001>;
pub type U1002 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1002 = PInt<U1002>; pub type N1002 = NInt<U1002>;
```

```
pub type U1003 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1003 = PInt<U1003>; pub type N1003 = NInt<U1003>;
pub type U1004 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1004 = PInt<U1004>; pub type N1004 = NInt<U1004>;
pub type U1005 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1005 = PInt<U1005>; pub type N1005 = NInt<U1005>;
pub type U1006 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1006 = PInt<U1006>; pub type N1006 = NInt<U1006>;
pub type U1007 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1007 = PInt<U1007>; pub type N1007 = NInt<U1007>;
pub type U1008 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1008 = PInt<U1008>; pub type N1008 = NInt<U1008>;
pub type U1009 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1009 = PInt<U1009>; pub type N1009 = NInt<U1009>;
pub type U1010 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1010 = PInt<U1010>; pub type N1010 = NInt<U1010>;
pub type U1011 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1011 = PInt<U1011>; pub type N1011 = NInt<U1011>;
pub type U1012 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1012 = PInt<U1012>; pub type N1012 = NInt<U1012>;
pub type U1013 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1013 = PInt<U1013>; pub type N1013 = NInt<U1013>;
pub type U1014 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1014 = PInt<U1014>; pub type N1014 = NInt<U1014>;
pub type U1015 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1015 = PInt<U1015>; pub type N1015 = NInt<U1015>;
pub type U1016 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1016 = PInt<U1016>; pub type N1016 = NInt<U1016>;
pub type U1017 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1017 = PInt<U1017>; pub type N1017 = NInt<U1017>;
pub type U1018 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1018 = PInt<U1018>; pub type N1018 = NInt<U1018>;
pub type U1019 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1019 = PInt<U1019>; pub type N1019 = NInt<U1019>;
pub type U1020 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1020 = PInt<U1020>; pub type N1020 = NInt<U1020>;
pub type U1021 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1021 = PInt<U1021>; pub type N1021 = NInt<U1021>;
pub type U1022 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1022 = PInt<U1022>; pub type N1022 = NInt<U1022>;
pub type U1023 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1023 = PInt<U1023>; pub type N1023 = NInt<U1023>;
pub type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1024 = PInt<U1024>; pub type N1024 = NInt<U1024>;
pub type U2048 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2048 = PInt<U2048>; pub type N2048 = NInt<U2048>;
pub type U4096 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4096 = PInt<U4096>; pub type N4096 = NInt<U4096>;
pub type U8192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P8192 = PInt<U8192>; pub type N8192 = NInt<U8192>;
pub type U16384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P16384 = PInt<U16384>; pub type N16384 = NInt<U16384>;
```

```
pub type U32768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P32768 = PInt<U32768>; pub type N32768 = NInt<U32768>;
pub type U65536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P65536 = PInt<U65536>; pub type N65536 = NInt<U65536>;
pub type U131072 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P131072 = PInt<U131072>; pub type N131072 = NInt<U131072>;
pub type U262144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P262144 = PInt<U262144>; pub type N262144 = NInt<U262144>;
pub type U524288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P524288 = PInt<U524288>; pub type N524288 = NInt<U524288>;
pub type U1048576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1048576 = PInt<U1048576>; pub type N1048576 = NInt<U1048576>;
pub type U2097152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P2097152 = PInt<U2097152>; pub type N2097152 = NInt<U2097152>;
pub type U4194304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P4194304 = PInt<U4194304>; pub type N4194304 = NInt<U4194304>;
pub type U8388608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P8388608 = PInt<U8388608>; pub type N8388608 = NInt<U8388608>;
pub type U16777216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P16777216 = PInt<U16777216>; pub type N16777216 = NInt<U167772
pub type U33554432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P33554432 = PInt<U33554432>; pub type N33554432 = NInt<U335544
pub type U67108864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P67108864 = PInt<U67108864>; pub type N67108864 = NInt<U671088
pub type U134217728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P134217728 = PInt<U134217728>; pub type N134217728 = NInt<U134
pub type U268435456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P268435456 = PInt<U268435456>; pub type N268435456 = NInt<U268
pub type U536870912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P536870912 = PInt<U536870912>; pub type N536870912 = NInt<U536
pub type U1073741824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1073741824 = PInt<U1073741824>; pub type N1073741824 = NInt<U
pub type U2147483648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P2147483648 = PInt<U2147483648>; pub type N2147483648 = NInt<U
pub type U4294967296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P4294967296 = PInt<U4294967296>; pub type N4294967296 = NInt<U
pub type U8589934592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P8589934592 = PInt<U8589934592>; pub type N8589934592 = NInt<U
pub type U17179869184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P17179869184 = PInt<U17179869184>; pub type N17179869184 = NIn
pub type U34359738368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P34359738368 = PInt<U34359738368>; pub type N34359738368 = NIn
pub type U68719476736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P68719476736 = PInt<U68719476736>; pub type N68719476736 = NIn
pub type U137438953472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P137438953472 = PInt<U137438953472>; pub type N137438953472 =
pub type U274877906944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P274877906944 = PInt<U274877906944>; pub type N274877906944 =
pub type U549755813888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P549755813888 = PInt<U549755813888>; pub type N549755813888 =
pub type U1099511627776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1099511627776 = PInt<U1099511627776>; pub type N1099511627776
```

```
pub type U2199023255552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P2199023255552 = PInt<U2199023255552>; pub type N2199023255552
pub type U4398046511104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P4398046511104 = PInt<U4398046511104>; pub type N4398046511104
pub type U8796093022208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P8796093022208 = PInt<U8796093022208>; pub type N8796093022208
pub type U17592186044416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P17592186044416 = PInt<U17592186044416>; pub type N1759218604
pub type U35184372088832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P35184372088832 = PInt<U35184372088832>; pub type N3518437208
pub type U70368744177664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P70368744177664 = PInt<U70368744177664>; pub type N7036874417
pub type U140737488355328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P140737488355328 = PInt<U140737488355328>; pub type N14073748
pub type U281474976710656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P281474976710656 = PInt<U281474976710656>; pub type N28147497
pub type U562949953421312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P562949953421312 = PInt<U562949953421312>; pub type N56294995
pub type U1125899906842624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P1125899906842624 = PInt<U1125899906842624>; pub type N112589
pub type U2251799813685248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P2251799813685248 = PInt<U2251799813685248>; pub type N225179
pub type U4503599627370496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P4503599627370496 = PInt<U4503599627370496>; pub type N450359
pub type U9007199254740992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P9007199254740992 = PInt<U9007199254740992>; pub type N900719
pub type U18014398509481984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P18014398509481984 = PInt<U18014398509481984>; pub type N18014
pub type U36028797018963968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P36028797018963968 = PInt<U36028797018963968>; pub type N36028
pub type U72057594037927936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P72057594037927936 = PInt<U72057594037927936>; pub type N72057
pub type U144115188075855872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P144115188075855872 = PInt<U144115188075855872>; pub type N144
pub type U288230376151711744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P288230376151711744 = PInt<U288230376151711744>; pub type N288
pub type U576460752303423488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P576460752303423488 = PInt<U576460752303423488>; pub type N576
pub type U1152921504606846976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1152921504606846976 = PInt<U1152921504606846976>; pub type N1
pub type U2305843009213693952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2305843009213693952 = PInt<U2305843009213693952>; pub type N2
pub type U4611686018427387904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4611686018427387904 = PInt<U4611686018427387904>; pub type N4
pub type U9223372036854775808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type U10000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P10000 = PInt<U10000>; pub type N10000 = NInt<U10000>;
pub type U100000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P100000 = PInt<U100000>; pub type N100000 = NInt<U100000>;
pub type U1000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1000000 = PInt<U1000000>; pub type N1000000 = NInt<U1000000>;
pub type U10000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
```

```
    pub type P10000000 = PInt<U10000000>; pub type N10000000 = NInt<U100000
    pub type U100000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P100000000 = PInt<U100000000>; pub type N100000000 = NInt<U100
    pub type U1000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P1000000000 = PInt<U1000000000>; pub type N1000000000 = NInt<U
    pub type U10000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P10000000000 = PInt<U10000000000>; pub type N10000000000 = NIn
    pub type U100000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P100000000000 = PInt<U100000000000>; pub type N100000000000 =
    pub type U1000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P1000000000000 = PInt<U1000000000000>; pub type N1000000000000
    pub type U10000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P10000000000000 = PInt<U10000000000000>; pub type N10000000000
    pub type U100000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P100000000000000 = PInt<U100000000000000>; pub type N100000000
    pub type U1000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P1000000000000000 = PInt<U1000000000000000>; pub type N1000000
    pub type U10000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P10000000000000000 = PInt<U10000000000000000>; pub type N10000
    pub type U100000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P100000000000000000 = PInt<U100000000000000000>; pub type N100
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P1000000000000000000 = PInt<U1000000000000000000>; pub type N1
    pub type U10000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
}
```

File: ./target/aarch64-apple-darwin/release/build/typenum-5ef8f1658

```
extern crate typenum;

use std::ops::*;
use std::cmp::Ordering;
use typenum::*;

#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_0() {
```

```
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitOrU0 = <<A as BitOr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitOrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitXorU0 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitXorU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU0 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU0 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0AddU0 = <<A as Add<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0AddU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MaxU0 = <<A as Max<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MaxU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0GcdU0 = <<A as Gcd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0GcdU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Sub_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0SubU0 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0SubU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_0() {
    type A = UTerm;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_0() {
    type A = UTerm;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U0CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU1 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitXorU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU1 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU1 = <<A as Add<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U0AddU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU1 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU1 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_Div_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU1 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU1 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U0CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU2 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitXorU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU2 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU2 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0AddU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU2 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_2() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU2 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU2 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_BitXor_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU3 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitXorU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU3 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU3 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0AddU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU3 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU3 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU3 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_4() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU4 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitXorU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU4 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU4 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0AddU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU4 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU4 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_4() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU4 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU5 = <<A as BitXor<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U0BitXorU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU5 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU5 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0AddU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU5 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_Max_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU5 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU5 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU0 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU0 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitXorU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShlU0 = <<A as Shl<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShlU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShrU0 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_0() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU0 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1AddU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU0 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU0 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U1SubU0 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1SubU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U1CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_BitOr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitXorU1 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitXorU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU1 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1ShlU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
```

```
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU1 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1AddU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1SubU1 = <<A as Sub<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U1SubU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU1 = <<A as Mul<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MulU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1DivU1 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1DivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_PartialDiv_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U1PartialDivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Pow_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU1 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U1BitXorU2 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitXorU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU2 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1ShlU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU2 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1AddU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU2 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU2 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MulU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_2() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU2 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;
```

```
    assert_eq!(<U1BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU3 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1BitXorU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU3 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U1ShlU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU3 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1AddU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Min_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU3 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU3 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MulU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU3 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU4 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitXorU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU4 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U1ShlU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_4() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU4 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1AddU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU4 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U1MulU4 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MulU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU4 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_BitAnd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU5 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1BitXorU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U1ShlU5 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U1ShlU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU5 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U1AddU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU5 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U1GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU5 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MulU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU5 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU5 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Cmp_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU0 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU0 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitXorU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2ShlU0 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShlU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShrU0 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU0 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2AddU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU0 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU0 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2SubU0 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2SubU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_0() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U2CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU1 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitXorU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU1 = <<A as Shl<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U2ShlU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU1 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2AddU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU1 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2SubU1 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2SubU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU1 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MulU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2DivU1 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2DivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U2PartialDivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU1 = <<A as Pow<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2PowU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitXorU2 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitXorU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU2 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2ShlU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_2() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU2 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2AddU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U2SubU2 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2SubU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU2 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MulU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U2PartialDivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU2 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2PowU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU3 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU3 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2BitXorU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU3 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2ShlU3 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU3 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2AddU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU3 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2MinU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU3 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2MulU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU3 = <<A as Pow<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2PowU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2BitOrU4 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitOrU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU4 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitXorU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU4 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2ShlU4 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU4 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2AddU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU4 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU4 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU4 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2MulU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_4() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU4 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU4 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2PowU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U2BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU5 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitXorU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU5 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U2ShlU5 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU5 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2AddU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU5 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU5 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U2MulU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU5 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2PowU5 = <<A as Pow<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2PowU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU5 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU0 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU0 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitXorU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShlU0 = <<A as Shl<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShlU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_0() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU0 = <<A as Shr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU0 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3AddU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU0 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3GcdU0 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU0 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3SubU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U3CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU1 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitXorU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU1 = <<A as Shl<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3ShlU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU1 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3AddU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU1 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U3GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3SubU1 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3SubU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU1 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MulU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU1 = <<A as Div<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3DivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U3>>::Outpu

    assert_eq!(<U3PartialDivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU1 = <<A as Pow<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3PowU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU2 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitXorU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU2 = <<A as Shl<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3ShlU2 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU2 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3AddU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU2 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU2 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3SubU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_2() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU2 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3MulU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU2 = <<A as Pow<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3PowU2 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU3 = <<A as BitAnd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitAndU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitXorU3 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitXorU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU3 = <<A as Shl<B>>::Output as Same<U24>>::Output;

    assert_eq!(<U3ShlU3 as Unsigned>::to_u64(), <U24 as Unsigned>::to_u64()
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU3 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3AddU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3SubU3 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3SubU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU3 = <<A as Mul<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3MulU3 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U3PartialDivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U27 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU3 = <<A as Pow<B>>::Output as Same<U27>>::Output;

    assert_eq!(<U3PowU3 as Unsigned>::to_u64(), <U27 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_4() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU4 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU4 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitXorU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U48 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU4 = <<A as Shl<B>>::Output as Same<U48>>::Output;

    assert_eq!(<U3ShlU4 as Unsigned>::to_u64(), <U48 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3AddU4 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3AddU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU4 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU4 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3MulU4 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU4 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U3PowU4 = <<A as Pow<B>>::Output as Same<U81>>::Output;

    assert_eq!(<U3PowU4 as Unsigned>::to_u64(), <U81 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU5 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3BitXorU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU5 = <<A as Shl<B>>::Output as Same<U96>>::Output;

    assert_eq!(<U3ShlU5 as Unsigned>::to_u64(), <U96 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU5 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U3AddU5 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU5 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU5 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U3MulU5 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU5 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U3PowU5 = <<A as Pow<B>>::Output as Same<U243>>::Output;

    assert_eq!(<U3PowU5 as Unsigned>::to_u64(), <U243 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU0 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU0 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitXorU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU0 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShlU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU0 = <<A as Shr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU0 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4AddU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU0 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_0() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU0 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU0 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4SubU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U4CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU1 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitXorU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU1 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4ShlU1 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU1 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4AddU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU1 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU1 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4SubU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU1 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MulU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU1 = <<A as Div<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4DivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U4RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U4>>::Outpu

    assert_eq!(<U4PartialDivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU1 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4PowU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_2() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU2 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitOrU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU2 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitXorU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU2 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4ShlU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4AddU2 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4AddU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU2 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU2 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4SubU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU2 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4MulU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U4PartialDivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_2() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU2 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4PowU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU3 = <<A as BitXor<B>>::Output as Same<U7>>::Output;
```

```
    assert_eq!(<U4BitXorU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU3 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U4ShlU3 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU3 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4AddU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU3 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU3 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4SubU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U12 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU3 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U4MulU3 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4PowU3 = <<A as Pow<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4PowU3 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitXorU4 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitXorU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU4 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4ShlU4 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_4() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU4 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4AddU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4SubU4 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4SubU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU4 = <<A as Mul<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4MulU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U4PartialDivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0

    #[allow(non_camel_case_types)]
    type U4PowU4 = <<A as Pow<B>>::Output as Same<U256>>::Output;

    assert_eq!(<U4PowU4 as Unsigned>::to_u64(), <U256 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU5 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU5 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4BitXorU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0

    #[allow(non_camel_case_types)]
    type U4ShlU5 = <<A as Shl<B>>::Output as Same<U128>>::Output;

    assert_eq!(<U4ShlU5 as Unsigned>::to_u64(), <U128 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU5 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U4AddU5 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU5 = <<A as Min<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4MinU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU5 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U4MulU5 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4RemU5 = <<A as Rem<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4RemU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTe

    #[allow(non_camel_case_types)]
    type U4PowU5 = <<A as Pow<B>>::Output as Same<U1024>>::Output;

    assert_eq!(<U4PowU5 as Unsigned>::to_u64(), <U1024 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitOrU0 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU0 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitXorU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShlU0 = <<A as Shl<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShlU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU0 = <<A as Shr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU0 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5AddU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU0 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU0 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU0 = <<A as Sub<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5SubU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_0() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U5CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U5BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU1 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitXorU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU1 = <<A as Shl<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5ShlU1 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU1 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5AddU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU1 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU1 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5SubU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
```

```
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU1 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MulU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU1 = <<A as Div<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5DivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U5>>::Outpu

    assert_eq!(<U5PartialDivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU1 = <<A as Pow<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U5PowU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU2 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU2 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitXorU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_2() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU2 = <<A as Shl<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5ShlU2 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU2 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5AddU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5MaxU2 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU2 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5SubU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU2 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5MulU2 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU2 = <<A as Pow<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5PowU2 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU3 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5BitXorU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU3 = <<A as Shl<B>>::Output as Same<U40>>::Output;

    assert_eq!(<U5ShlU3 as Unsigned>::to_u64(), <U40 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU3 = <<A as Add<B>>::Output as Same<U8>>::Output;
```

```
    assert_eq!(<U5AddU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU3 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU3 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5SubU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU3 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U5MulU3 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U5PowU3 = <<A as Pow<B>>::Output as Same<U125>>::Output;

    assert_eq!(<U5PowU3 as Unsigned>::to_u64(), <U125 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU4 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitXorU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU4 = <<A as Shl<B>>::Output as Same<U80>>::Output;

    assert_eq!(<U5ShlU4 as Unsigned>::to_u64(), <U80 as Unsigned>::to_u64()
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU4 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U5AddU4 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU4 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_4() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU4 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5SubU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU4 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5MulU4 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type U5PowU4 = <<A as Pow<B>>::Output as Same<U625>>::Output;

    assert_eq!(<U5PowU4 as Unsigned>::to_u64(), <U625 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU5 = <<A as BitAnd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitAndU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitXorU5 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitXorU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1

    #[allow(non_camel_case_types)]
    type U5ShlU5 = <<A as Shl<B>>::Output as Same<U160>>::Output;

    assert_eq!(<U5ShlU5 as Unsigned>::to_u64(), <U160 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU5 = <<A as Add<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5AddU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU5 = <<A as Min<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MinU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5SubU5 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5SubU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU5 = <<A as Mul<B>>::Output as Same<U25>>::Output;
```

```
    assert_eq!(<U5MulU5 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU5 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U5PartialDivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type U5PowU5 = <<A as Pow<B>>::Output as Same<U3125>>::Output;

    assert_eq!(<U5PowU5 as Unsigned>::to_u64(), <U3125 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN5 = <<A as Add<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5AddN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5SubN5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5SubN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulN5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5MulN5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN5 = <<A as Max<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MaxN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N5PartialDivN5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN4 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5AddN4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5SubN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulN4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N5MulN4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N5DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN3 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5AddN3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5SubN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N3() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N5MulN3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN2 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5AddN2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5SubN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulN2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N5MulN2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN1 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5AddN1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5SubN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MulN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N1() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5DivN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<N5PartialDivN1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Add_0 = <<A as Add<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5Add_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Sub_0 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Sub_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Min_0 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Min_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N5Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5AddP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N5SubP1 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5SubP1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MulP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5DivP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<N5PartialDivP1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PowP1 = <<A as Pow<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5PowP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P1() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5AddP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP2 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5SubP2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulP2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5MulP2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP2 = <<A as Min<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5MinP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5AddP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP3 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5SubP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MulP3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N5MulP3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N5MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type N5PowP3 = <<A as Pow<B>>::Output as Same<N125>>::Output;

    assert_eq!(<N5PowP3 as Integer>::to_i64(), <N125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5AddP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP4 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5SubP4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulP4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N5MulP4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N5MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type N5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<N5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P5() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5AddP5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5AddP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP5 = <<A as Sub<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5SubP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulP5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<N5MulP5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N5PartialDivP5 as Integer>::to_i64(), <N1 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N5PowP5 = <<A as Pow<B>>::Output as Same<N3125>>::Output;

    assert_eq!(<N5PowP5 as Integer>::to_i64(), <N3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN5 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4AddN5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN5 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4SubN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N4MulN5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN5 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemN5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN4 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4AddN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4SubN4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4SubN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N4() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4MulN4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N4DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N4PartialDivN4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN3 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4AddN3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4SubN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N4MulN3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN2 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4AddN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubN2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4SubN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N4MulN2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N2() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N4PartialDivN2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN1 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4AddN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4SubN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MulN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4DivN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<N4PartialDivN1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Add_0 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Add_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Sub_0 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Sub_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Min_0 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Min_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp__0() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N4Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4AddP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP1 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4SubP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MulP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP1 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4MinP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4DivP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<N4PartialDivP1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PowP1 = <<A as Pow<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4PowP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddP2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4AddP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4SubP2 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4SubP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4MulP2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N4PartialDivP2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P2() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4AddP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP3 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4SubP3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N4MulP3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP3 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4MinP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N4MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N64 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>
    #[allow(non_camel_case_types)]
    type N4PowP3 = <<A as Pow<B>>::Output as Same<N64>>::Output;
    assert_eq!(<N4PowP3 as Integer>::to_i64(), <N64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    #[allow(non_camel_case_types)]
    type N4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
    #[allow(non_camel_case_types)]
    type N4AddP4 = <<A as Add<B>>::Output as Same<_0>>::Output;
    assert_eq!(<N4AddP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
    #[allow(non_camel_case_types)]
    type N4SubP4 = <<A as Sub<B>>::Output as Same<N8>>::Output;
    assert_eq!(<N4SubP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type N4MulP4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<N4MulP4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N4PartialDivP4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type N4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<N4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP5 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4AddP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP5 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4SubP5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N4MulP5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP5 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<N4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemP5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1024 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N4PowP5 = <<A as Pow<B>>::Output as Same<N1024>>::Output;

    assert_eq!(<N4PowP5 as Integer>::to_i64(), <N1024 as Integer>::to_i64()
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN5 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3AddN5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN5 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3SubN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N3MulN5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN5 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN4 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3AddN4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3SubN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N3MulN4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN4 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN4 = <<A as Rem<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3RemN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN3 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3AddN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3SubN3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3SubN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3MulN3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N3() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N3PartialDivN3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN2 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3AddN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3SubN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N3MulN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3AddN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3SubN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MulN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N1() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3DivN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<N3PartialDivN1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Add_0 = <<A as Add<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3Add_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Sub_0 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Sub_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Min_0 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Min_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N3Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3AddP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N3SubP1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3SubP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MulP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3DivP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<N3PartialDivP1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP1 = <<A as Pow<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3PowP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P1() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3AddP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP2 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3SubP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3MulP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP2 = <<A as Min<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3MinP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3AddP3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3AddP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP3 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3SubP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MulP3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N3MulP3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N3PartialDivP3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N27 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP3 = <<A as Pow<B>>::Output as Same<N27>>::Output;

    assert_eq!(<N3PowP3 as Integer>::to_i64(), <N27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3AddP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP4 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3SubP4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N3MulP4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP4 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<N3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<N3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP5 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3AddP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP5 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3SubP5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N3MulP5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinP5 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N243 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type N3PowP5 = <<A as Pow<B>>::Output as Same<N243>>::Output;

    assert_eq!(<N3PowP5 as Integer>::to_i64(), <N243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN5 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2AddN5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN5 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2SubN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N2MulN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN5 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN4 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2AddN4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubN4 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2SubN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N4() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N2MulN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN4 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN3 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2AddN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2SubN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N2MulN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN3 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN2 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2AddN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2SubN2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2SubN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MulN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N2() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N2PartialDivN2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN1 = <<A as Add<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N2AddN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2SubN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MulN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivN1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2DivN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N2PartialDivN1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Add_0 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Add_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Sub_0 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Sub_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Min_0 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Min_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N2Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2AddP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2SubP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MulP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N2MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivP1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2DivP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N2PartialDivP1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP1 = <<A as Pow<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2PowP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2AddP2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2AddP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP2 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2SubP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MulP2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MulP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N2PartialDivP2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2AddP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP3 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2SubP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2MulP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP3 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<N2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP3 = <<A as Pow<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2PowP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddP4 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2AddP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP4 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2SubP4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2MulP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MinP4 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP5 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2AddP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP5 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2SubP5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N2MulP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP5 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N32 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type N2PowP5 = <<A as Pow<B>>::Output as Same<N32>>::Output;

    assert_eq!(<N2PowP5 as Integer>::to_i64(), <N32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN5 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1AddN5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N5() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN5 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1SubN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MulN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN5 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Add_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN4 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1AddN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN4 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1SubN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MulN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN4 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN3 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1AddN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1SubN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MulN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N3() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN3 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1AddN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1SubN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Mul_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MulN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN2 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1AddN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1SubN1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1SubN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MulN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N1() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivN1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1DivN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N1PartialDivN1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1PowN1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Add_0 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Add_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Sub_0 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Sub_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Min__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Min_0 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Min_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N1Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1AddP1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1AddP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1SubP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MulP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivP1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1DivP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_P1() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N1PartialDivP1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1AddP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP2 = <<A as Sub<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N1SubP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MulP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP2 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Div_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N1AddP3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1AddP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP3 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1SubP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MulP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP3 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P3() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP4 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1AddP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP4 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1SubP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MulP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP4 = <<A as Min<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N1MinP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Pow_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP5 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1AddP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP5 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1SubP5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1MulP5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MulP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP5 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN5 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0AddN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN5 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0SubN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN5 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<_0GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N4() {
```

```
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN4 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0AddN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN4 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0SubN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0MaxN4 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN4 as Integer>::to_i64(), <_0 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN3 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0AddN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN3 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0SubN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN3 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0AddN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0SubN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N2() {
```

```
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN2 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0AddN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Sub_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0SubN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN1 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN1 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Add_0 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Add_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Sub_0 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Sub_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max__0() {
```

```
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Gcd_0 = <<A as Gcd<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Gcd_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow__0() {
    type A = Z0;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp__0() {
    type A = Z0;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type _0Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<_0Cmp_0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP1 = <<A as Add<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<_0AddP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0SubP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP1 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Gcd_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP1 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0AddP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0SubP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP2 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P2() {
```

```
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP2 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP3 = <<A as Add<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<_0AddP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP3 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0SubP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP3 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Gcd_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP3 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP4 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0AddP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP4 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0SubP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP4 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P4() {
```

```
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP4 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP5 = <<A as Add<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<_0AddP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP5 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0SubP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP5 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Gcd_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP5 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN5 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1AddN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN5 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1SubN5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MulN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN5 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N5() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN4 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1AddN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN4 = <<A as Sub<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P1SubN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MulN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN4 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Div_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P1AddN3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1AddN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN3 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1SubN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MulN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN3 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N3() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1AddN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1SubN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MulN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<P1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN2 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Pow_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1AddN1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1AddN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1SubN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1MulN1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MulN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivN1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1DivN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P1PartialDivN1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Add_0 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Add_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Sub_0 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Sub_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Max_0 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1Max_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P1Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1AddP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P1() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1SubP1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1SubP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MulP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivP1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1DivP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P1PartialDivP1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1AddP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1SubP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MulP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP2 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP3 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1AddP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1SubP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P3() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MulP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP3 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type P1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP4 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1AddP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Sub_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP4 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1SubP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MulP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP4 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP4 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP5 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1AddP5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP5 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1SubP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MulP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP5 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P5() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1PowP5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN5 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2AddN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN5 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2SubN5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P2MulN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN5 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddN4 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2AddN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN4 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2SubN4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P2MulN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN4 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N4() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2AddN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN3 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2SubN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN3 = <<A as Mul<B>>::Output as Same<N6>>::Output;
```

```
    assert_eq!(<P2MulN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN3 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2AddN2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2AddN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN2 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2SubN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2MulN2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MulN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P2PartialDivN2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2AddN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2SubN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MulN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivN1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2DivN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P2PartialDivN1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add__0() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Add_0 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Add_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Sub_0 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Sub_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2Max_0 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Max_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P2Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2AddP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2SubP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MulP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivP1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2DivP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P2PartialDivP1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP1 = <<A as Pow<B>>::Output as Same<P2>>::Output;
```

```
    assert_eq!(<P2PowP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP2 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2AddP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2SubP2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2SubP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MulP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P2() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type P2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P2PartialDivP2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP3 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2AddP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2SubP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2MulP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP3 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP3 = <<A as Pow<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2PowP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP3 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP4 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2AddP4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubP4 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2SubP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2MulP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP4 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P4() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type P2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP5 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2AddP5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP5 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2SubP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P2MulP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP5 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P32 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP5 = <<A as Pow<B>>::Output as Same<P32>>::Output;

    assert_eq!(<P2PowP5 as Integer>::to_i64(), <P32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN5 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3AddN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN5 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3SubN5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P3MulN5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN5 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N5() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3AddN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN4 = <<A as Sub<B>>::Output as Same<P7>>::Output;
```

```
    assert_eq!(<P3SubN4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P3MulN4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN4 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3AddN3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3AddN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P3SubN3 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3SubN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<P3MulN3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P3PartialDivN3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3AddN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN2 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3SubN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P3MulN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN2 = <<A as Max<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<P3MaxN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N1() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3AddN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3SubN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MulN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3MaxN1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3DivN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<P3PartialDivN1 as Integer>::to_i64(), <N3 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Add_0 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Add_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Sub_0 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Sub_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Max_0 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Max_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P3Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3AddP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3SubP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MulP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P1() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3DivP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<P3PartialDivP1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP1 = <<A as Pow<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3PowP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP2 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3AddP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3SubP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3MulP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP3 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3AddP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3SubP3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3SubP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3MulP3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P3() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P3PartialDivP3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P27 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>
```

```
    #[allow(non_camel_case_types)]
    type P3PowP3 = <<A as Pow<B>>::Output as Same<P27>>::Output;

    assert_eq!(<P3PowP3 as Integer>::to_i64(), <P27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP4 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3AddP4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3SubP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P3MulP4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP4 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<P3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP5 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3AddP5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP5 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3SubP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P3MulP5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP5 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P5() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P243 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type P3PowP5 = <<A as Pow<B>>::Output as Same<P243>>::Output;

    assert_eq!(<P3PowP5 as Integer>::to_i64(), <P243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN5 = <<A as Add<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P4AddN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN5 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4SubN5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P4MulN5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN5 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemN5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type P4AddN4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4AddN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN4 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4SubN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<P4MulN4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P4PartialDivN4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N4() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4AddN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN3 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4SubN3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P4MulN3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<P4MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddN2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4AddN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN2 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4SubN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P4MulN2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P4PartialDivN2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4AddN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN1 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4SubN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MulN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN1 = <<A as Div<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<P4DivN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<P4PartialDivN1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Add_0 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Add_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub__0() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Sub_0 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Sub_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Max_0 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Max_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P4Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP1 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4AddP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4SubP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MulP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4DivP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<P4PartialDivP1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PowP1 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4PowP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP1 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP2 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4AddP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubP2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4SubP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4MulP2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P2() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P4PartialDivP2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP3 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4AddP3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4SubP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P4MulP3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P64 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP3 = <<A as Pow<B>>::Output as Same<P64>>::Output;

    assert_eq!(<P4PowP3 as Integer>::to_i64(), <P64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP4 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4AddP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4SubP4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4SubP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4MulP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P4() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P4PartialDivP4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1
```

```
    #[allow(non_camel_case_types)]
    type P4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<P4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP5 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4AddP5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP5 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4SubP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P4MulP5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP5 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemP5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1024 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P4PowP5 = <<A as Pow<B>>::Output as Same<P1024>>::Output;

    assert_eq!(<P4PowP5 as Integer>::to_i64(), <P1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5AddN5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5AddN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN5 = <<A as Sub<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5SubN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulN5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<P5MulN5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N5() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P5PartialDivN5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN4 = <<A as Add<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P5AddN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN4 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5SubN4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulN4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P5MulN4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P5MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P5AddN3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5AddN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN3 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5SubN3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P5MulN3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P5MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5AddN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN2 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5SubN2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulN2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P5MulN2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN2 = <<A as Max<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P5MaxN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N1() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5AddN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN1 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5SubN1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MulN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5MaxN1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5DivN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<P5PartialDivN1 as Integer>::to_i64(), <N5 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Add_0 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Add_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Sub_0 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Sub_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Max_0 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Max_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P5Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP1 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5AddP1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5SubP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MulP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P1() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5DivP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<P5PartialDivP1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PowP1 = <<A as Pow<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5PowP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP2 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5AddP2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5SubP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulP2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5MulP2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP3 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5AddP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5SubP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P5MulP3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P3() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1
```

```
    #[allow(non_camel_case_types)]
    type P5PowP3 = <<A as Pow<B>>::Output as Same<P125>>::Output;

    assert_eq!(<P5PowP3 as Integer>::to_i64(), <P125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP4 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5AddP4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5SubP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulP4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P5MulP4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type P5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<P5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP5 = <<A as Add<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5AddP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5SubP5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5SubP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulP5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5MulP5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP5 = <<A as Min<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MinP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P5() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P5PartialDivP5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P5PowP5 = <<A as Pow<B>>::Output as Same<P3125>>::Output;

    assert_eq!(<P5PowP5 as Integer>::to_i64(), <P3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN5 = <<A as Neg>::Output as Same<P5>>::Output;
    assert_eq!(<NegN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN4 = <<A as Neg>::Output as Same<P4>>::Output;
    assert_eq!(<NegN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN3 = <<A as Neg>::Output as Same<P3>>::Output;
    assert_eq!(<NegN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN2 = <<A as Neg>::Output as Same<P2>>::Output;
    assert_eq!(<NegN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Neg() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegN1 = <<A as Neg>::Output as Same<P1>>::Output;
    assert_eq!(<NegN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Abs() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Neg() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Neg_0 = <<A as Neg>::Output as Same<_0>>::Output;
    assert_eq!(<Neg_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Abs() {
```

```
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Abs_0 = <<A as Abs>::Output as Same<_0>>::Output;
    assert_eq!(<Abs_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Neg() {
    type A = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegP1 = <<A as Neg>::Output as Same<N1>>::Output;
    assert_eq!(<NegP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Abs() {
    type A = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP2 = <<A as Neg>::Output as Same<N2>>::Output;
    assert_eq!(<NegP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type NegP3 = <<A as Neg>::Output as Same<N3>>::Output;
    assert_eq!(<NegP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP4 = <<A as Neg>::Output as Same<N4>>::Output;
    assert_eq!(<NegP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP5 = <<A as Neg>::Output as Same<N5>>::Output;
    assert_eq!(<NegP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type AbsP5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

File: ./target/aarch64-apple-darwin/release/build/typenum-5ef8f1658

```
/**
Convenient type operations.

Any types representing values must be able to be expressed as `ident`s. Tha
in scope.

For example, `P5` is okay, but `typenum::P5` is not.

You may combine operators arbitrarily, although doing so excessively may re
recursion limit.

# Example
```rust
#![recursion_limit="128"]
#[macro_use] extern crate typenum;
use typenum::consts::*;

fn main() {
    assert_type!(
        op!(min((P1 - P2) * (N3 + N7), P5 * (P3 + P4)) == P10)
    );
}
```
Operators are evaluated based on the operator precedence outlined
[here](https://doc.rust-lang.org/reference.html#operator-precedence).

The full list of supported operators and functions is as follows:

`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, `==`, `!=`, `<=`, `>=`,

They all expand to type aliases defined in the `operator_aliases` module. H
including examples:

Operator `*`. Expands to `Prod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 * P3), P6);
# }
```
```

Operator `/`. Expands to `Quot`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 / P2), P3);
# }
```

Operator `%`. Expands to `Mod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 % P3), P2);
# }
```

Operator `+`. Expands to `Sum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 + P3), P5);
# }
```

Operator `-`. Expands to `Diff`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 - P3), N1);
# }
```

Operator `<<`. Expands to `Shleft`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
```

```
assert_type_eq!(op!(U1 << U5), U32);
# }
```

Operator `>>`. Expands to `Shright`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U32 >> U5), U1);
# }
```

Operator `&`. Expands to `And`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 & U3), U1);
# }
```

Operator `^`. Expands to `Xor`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 ^ U3), U6);
# }
```

Operator `|`. Expands to `Or`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 | U3), U7);
# }
```

Operator `==`. Expands to `Eq`.

```rust
```

```
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 == P3 + P2), True);
# }
```

Operator `!=`. Expands to `NotEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 != P3 + P2), False);
# }
```

Operator `<=`. Expands to `LeEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 <= P3 + P2), False);
# }
```

Operator `>=`. Expands to `GrEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 >= P3 + P2), True);
# }
```

Operator `<`. Expands to `Le`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P4 < P3 + P2), True);
# }
```

Operator `>`. Expands to `Gr`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 < P3 + P2), False);
# }
```

Operator `cmp`. Expands to `Compare`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cmp(P2, P3)), Less);
# }
```

Operator `sqr`. Expands to `Square`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqr(P2)), P4);
# }
```

Operator `sqrt`. Expands to `Sqrt`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqrt(U9)), U3);
# }
```

Operator `abs`. Expands to `AbsVal`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(abs(N2)), P2);
# }
```

```

---
Operator `cube`. Expands to `Cube`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cube(P2)), P8);
# }
```

Operator `pow`. Expands to `Exp`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(pow(P2, P3)), P8);
# }
```

Operator `min`. Expands to `Minimum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(min(P2, P3)), P2);
# }
```

Operator `max`. Expands to `Maximum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(max(P2, P3)), P3);
# }
```

Operator `log2`. Expands to `Log2`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
```
```

```
# fn main() {
assert_type_eq!(op!(log2(U9)), U3);
# }
```

Operator `gcd`. Expands to `Gcf`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(gcd(U9, U21)), U3);
# }
```

*/
#[macro_export(local_inner_macros)]
macro_rules! op {
    ($($tail:tt)*) => (__op_internal__!($($tail)*));
}

    #[doc(hidden)]
    #[macro_export(local_inner_macros)]
    macro_rules! __op_internal__ {

(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cmp $($tail:tt)
    __op_internal__!(@stack[Compare, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqr $($tail:tt)
    __op_internal__!(@stack[Square, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqrt $($tail:tt
    __op_internal__!(@stack[Sqrt, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: abs $($tail:tt)
    __op_internal__!(@stack[AbsVal, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cube $($tail:tt
    __op_internal__!(@stack[Cube, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: pow $($tail:tt)
    __op_internal__!(@stack[Exp, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: min $($tail:tt)
    __op_internal__!(@stack[Minimum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: max $($tail:tt)
    __op_internal__!(@stack[Maximum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: log2 $($tail:tt
    __op_internal__!(@stack[Log2, $($stack,)*] @queue[$($queue,)*] @tail: $

```
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: gcd $($tail:tt)
    __op_internal__!(@stack[Gcf, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: , $($ta
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: *
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: *
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: *
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:tt)*)
    __op_internal__!(@stack[Prod, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: /
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: /
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: /
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:tt)*)
    __op_internal__!(@stack[Quot, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: %
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: %
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: %
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:tt)*)
    __op_internal__!(@stack[Mod, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: +
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: +
);
```

```
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: +
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: +
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: +
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:tt)*)
    __op_internal__!(@stack[Sum, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: -
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: -
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: -
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: -
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: -
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:tt)*)
    __op_internal__!(@stack[Diff, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <<
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <<
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail:tt)*
```

```
    __op_internal__!(@stack[Shleft, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >>
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >>
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail:tt)*
    __op_internal__!(@stack[Shright, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: &
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: &
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: &
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: &
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: &
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: &
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:tt)*)
    __op_internal__!(@stack[And, $($stack,)*] @queue[$($queue,)*] @tail: $(
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: ^
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: ^
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ^
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ^
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: ^
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ^
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ^
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:tt)*)
    __op_internal__!(@stack[Xor, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: |
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: |
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: |
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: |
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: |
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
```

```
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: |
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: |
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: | $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:tt)*)
    __op_internal__!(@stack[Or, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: =
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: =
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ==
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ==
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: =
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ==
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ==
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: ==
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: ==
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: =
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: =
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: ==
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: ==
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:tt)*
    __op_internal__!(@stack[Eq, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: !
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: !
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: !=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: !=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: !
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: !=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: !=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: !=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: !=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: !
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: !
```

```
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: !=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: !=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:tt)*
    __op_internal__!(@stack[NotEq, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: <=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: <=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
```

```
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: <=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: <=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:tt)*
    __op_internal__!(@stack[LeEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: >=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: >=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: >=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: >=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:tt)*
    __op_internal__!(@stack[GrEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: < $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: < $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: < $
```

```
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: < $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:tt)*)
    __op_internal__!(@stack[Le, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: > $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: > $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: > $
);
```

```
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: > $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:tt)*)
    __op_internal__!(@stack[Gr, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ( $($stuff:tt)*
 => (
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*]
                     @tail: $($stuff)* RParen $($tail)*)
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: RParen
    __op_internal__!(@rp3 @stack[$($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
 => (
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@rp3 @stack[Compare, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Compare, $($queue,)*] @tail
);
(@rp3 @stack[Square, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[Square, $($queue,)*] @tail:
);
(@rp3 @stack[Sqrt, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Sqrt, $($queue,)*] @tail: $
);
(@rp3 @stack[AbsVal, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[AbsVal, $($queue,)*] @tail:
);
(@rp3 @stack[Cube, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Cube, $($queue,)*] @tail: $
);
(@rp3 @stack[Exp, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Exp, $($queue,)*] @tail: $(
);
(@rp3 @stack[Minimum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Minimum, $($queue,)*] @tail
);
(@rp3 @stack[Maximum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Maximum, $($queue,)*] @tail
);
(@rp3 @stack[Log2, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Log2, $($queue,)*] @tail: $
);
(@rp3 @stack[Gcf, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Gcf, $($queue,)*] @tail: $(
);
(@rp3 @stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($tail:tt
    __op_internal__!(@stack[$($stack,)*] @queue[$($queue,)*] @tail: $($tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $num:ident $($t
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[$num, $($queue,)*] @tail: $
);
(@stack[] @queue[$($queue:ident,)*] @tail: ) => (
    __op_internal__!(@reverse[] @input: $($queue,)*)
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@reverse[$($revved:ident,)*] @input: $head:ident, $($tail:ident,)* ) => (
    __op_internal__!(@reverse[$head, $($revved,)*] @input: $($tail,)*)
);
(@reverse[$($revved:ident,)*] @input: ) => (
    __op_internal__!(@eval @stack[] @input[$($revved,)*])
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Prod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Prod<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Quot, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Quot<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Mod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Mod<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Sum, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Sum<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Diff, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Diff<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shleft, $($tail:ident,)*
    __op_internal__!(@eval @stack[$crate::Shleft<$b, $a>, $($stack,)*] @inp
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shright, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Shright<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[And, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::And<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Xor, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Xor<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Or, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Or<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Eq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Eq<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[NotEq, $($tail:ident,)*]
    __op_internal__!(@eval @stack[$crate::NotEq<$b, $a>, $($stack,)*] @inpu
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[LeEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::LeEq<$b, $a>, $($stack,)*] @input
```

```
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[GrEq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::GrEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Le, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Le<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gr, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Gr<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Compare, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Compare<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Exp, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Exp<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Minimum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Minimum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Maximum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Maximum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gcf, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Gcf<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Square, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Square<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Sqrt, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Sqrt<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[AbsVal, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::AbsVal<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Cube, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Cube<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Log2, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Log2<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$($stack:ty,)*] @input[$head:ident, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$head, $($stack,)*] @input[$($tail,)*])
);
(@eval @stack[$stack:ty,] @input[]) => (
    $stack
);
($($tail:tt)* ) => (
    __op_internal__!(@stack[] @queue[] @tail: $($tail)*)
);
}
```

File: ./target/release/build/typenum-346c6668244bcd95/out/consts.r

```
/**
Type aliases for many constants.

This file is generated by typenum's build script.

For unsigned integers, the format is `U` followed by the number. We define

- Numbers 0 through 1024
- Powers of 2 below `u64::MAX`
- Powers of 10 below `u64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm};

# #[allow(dead_code)]
type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

For positive signed integers, the format is `P` followed by the number and
signed integers it is `N` followed by the number. For the signed integer ze
`Z0`. We define aliases for

- Numbers -1024 through 1024
- Powers of 2 between `i64::MIN` and `i64::MAX`
- Powers of 10 between `i64::MIN` and `i64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm, PInt, NInt};

# #[allow(dead_code)]
type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
# #[allow(dead_code)]
type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```


Example
```rust
# #[allow(unused_imports)]
use typenum::{U0, U1, U2, U3, U4, U5, U6};
# #[allow(unused_imports)]
use typenum::{N3, N2, N1, Z0, P1, P2, P3};
# #[allow(unused_imports)]
use typenum::{U774, N17, N10000, P1024, P4096};
```
```

```
We also define the aliases `False` and `True` for `B0` and `B1`, respective
*/
#[allow(missing_docs)]
pub mod consts {
    use crate::uint::{UInt, UTerm};
    use crate::int::{PInt, NInt};

    pub use crate::bit::{B0, B1};
    pub use crate::int::Z0;

    pub type True = B1;
    pub type False = B0;
    pub type U0 = UTerm;
    pub type U1 = UInt<UTerm, B1>;
    pub type P1 = PInt<U1>; pub type N1 = NInt<U1>;
    pub type U2 = UInt<UInt<UTerm, B1>, B0>;
    pub type P2 = PInt<U2>; pub type N2 = NInt<U2>;
    pub type U3 = UInt<UInt<UTerm, B1>, B1>;
    pub type P3 = PInt<U3>; pub type N3 = NInt<U3>;
    pub type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    pub type P4 = PInt<U4>; pub type N4 = NInt<U4>;
    pub type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    pub type P5 = PInt<U5>; pub type N5 = NInt<U5>;
    pub type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
    pub type P6 = PInt<U6>; pub type N6 = NInt<U6>;
    pub type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
    pub type P7 = PInt<U7>; pub type N7 = NInt<U7>;
    pub type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
    pub type P8 = PInt<U8>; pub type N8 = NInt<U8>;
    pub type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;
    pub type P9 = PInt<U9>; pub type N9 = NInt<U9>;
    pub type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;
    pub type P10 = PInt<U10>; pub type N10 = NInt<U10>;
    pub type U11 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>;
    pub type P11 = PInt<U11>; pub type N11 = NInt<U11>;
    pub type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;
    pub type P12 = PInt<U12>; pub type N12 = NInt<U12>;
    pub type U13 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>;
    pub type P13 = PInt<U13>; pub type N13 = NInt<U13>;
    pub type U14 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>;
    pub type P14 = PInt<U14>; pub type N14 = NInt<U14>;
    pub type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;
    pub type P15 = PInt<U15>; pub type N15 = NInt<U15>;
    pub type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
    pub type P16 = PInt<U16>; pub type N16 = NInt<U16>;
    pub type U17 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>;
    pub type P17 = PInt<U17>; pub type N17 = NInt<U17>;
    pub type U18 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B0>;
    pub type P18 = PInt<U18>; pub type N18 = NInt<U18>;
    pub type U19 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B1>;
    pub type P19 = PInt<U19>; pub type N19 = NInt<U19>;
    pub type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;
```

```
pub type P20 = PInt<U20>; pub type N20 = NInt<U20>;
pub type U21 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B1>;
pub type P21 = PInt<U21>; pub type N21 = NInt<U21>;
pub type U22 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B0>;
pub type P22 = PInt<U22>; pub type N22 = NInt<U22>;
pub type U23 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B1>;
pub type P23 = PInt<U23>; pub type N23 = NInt<U23>;
pub type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;
pub type P24 = PInt<U24>; pub type N24 = NInt<U24>;
pub type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
pub type P25 = PInt<U25>; pub type N25 = NInt<U25>;
pub type U26 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B0>;
pub type P26 = PInt<U26>; pub type N26 = NInt<U26>;
pub type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;
pub type P27 = PInt<U27>; pub type N27 = NInt<U27>;
pub type U28 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B0>;
pub type P28 = PInt<U28>; pub type N28 = NInt<U28>;
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
pub type U35 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P35 = PInt<U35>; pub type N35 = NInt<U35>;
pub type U36 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P36 = PInt<U36>; pub type N36 = NInt<U36>;
pub type U37 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P37 = PInt<U37>; pub type N37 = NInt<U37>;
pub type U38 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P38 = PInt<U38>; pub type N38 = NInt<U38>;
pub type U39 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P39 = PInt<U39>; pub type N39 = NInt<U39>;
pub type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P40 = PInt<U40>; pub type N40 = NInt<U40>;
pub type U41 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P41 = PInt<U41>; pub type N41 = NInt<U41>;
pub type U42 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P42 = PInt<U42>; pub type N42 = NInt<U42>;
pub type U43 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P43 = PInt<U43>; pub type N43 = NInt<U43>;
pub type U44 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P44 = PInt<U44>; pub type N44 = NInt<U44>;
pub type U45 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P45 = PInt<U45>; pub type N45 = NInt<U45>;
pub type U46 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
```

```
pub type P46 = PInt<U46>; pub type N46 = NInt<U46>;
pub type U47 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P47 = PInt<U47>; pub type N47 = NInt<U47>;
pub type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P48 = PInt<U48>; pub type N48 = NInt<U48>;
pub type U49 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P49 = PInt<U49>; pub type N49 = NInt<U49>;
pub type U50 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P50 = PInt<U50>; pub type N50 = NInt<U50>;
pub type U51 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P51 = PInt<U51>; pub type N51 = NInt<U51>;
pub type U52 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P52 = PInt<U52>; pub type N52 = NInt<U52>;
pub type U53 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P53 = PInt<U53>; pub type N53 = NInt<U53>;
pub type U54 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P54 = PInt<U54>; pub type N54 = NInt<U54>;
pub type U55 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P55 = PInt<U55>; pub type N55 = NInt<U55>;
pub type U56 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P56 = PInt<U56>; pub type N56 = NInt<U56>;
pub type U57 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P57 = PInt<U57>; pub type N57 = NInt<U57>;
pub type U58 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P58 = PInt<U58>; pub type N58 = NInt<U58>;
pub type U59 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P59 = PInt<U59>; pub type N59 = NInt<U59>;
pub type U60 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P60 = PInt<U60>; pub type N60 = NInt<U60>;
pub type U61 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P61 = PInt<U61>; pub type N61 = NInt<U61>;
pub type U62 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P62 = PInt<U62>; pub type N62 = NInt<U62>;
pub type U63 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P63 = PInt<U63>; pub type N63 = NInt<U63>;
pub type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P64 = PInt<U64>; pub type N64 = NInt<U64>;
pub type U65 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P65 = PInt<U65>; pub type N65 = NInt<U65>;
pub type U66 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P66 = PInt<U66>; pub type N66 = NInt<U66>;
pub type U67 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P67 = PInt<U67>; pub type N67 = NInt<U67>;
pub type U68 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P68 = PInt<U68>; pub type N68 = NInt<U68>;
pub type U69 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P69 = PInt<U69>; pub type N69 = NInt<U69>;
pub type U70 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P70 = PInt<U70>; pub type N70 = NInt<U70>;
pub type U71 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P71 = PInt<U71>; pub type N71 = NInt<U71>;
pub type U72 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
```

```
pub type P72 = PInt<U72>; pub type N72 = NInt<U72>;
pub type U73 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P73 = PInt<U73>; pub type N73 = NInt<U73>;
pub type U74 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P74 = PInt<U74>; pub type N74 = NInt<U74>;
pub type U75 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P75 = PInt<U75>; pub type N75 = NInt<U75>;
pub type U76 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P76 = PInt<U76>; pub type N76 = NInt<U76>;
pub type U77 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P77 = PInt<U77>; pub type N77 = NInt<U77>;
pub type U78 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P78 = PInt<U78>; pub type N78 = NInt<U78>;
pub type U79 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P79 = PInt<U79>; pub type N79 = NInt<U79>;
pub type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P80 = PInt<U80>; pub type N80 = NInt<U80>;
pub type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P81 = PInt<U81>; pub type N81 = NInt<U81>;
pub type U82 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P82 = PInt<U82>; pub type N82 = NInt<U82>;
pub type U83 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P83 = PInt<U83>; pub type N83 = NInt<U83>;
pub type U84 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P84 = PInt<U84>; pub type N84 = NInt<U84>;
pub type U85 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P85 = PInt<U85>; pub type N85 = NInt<U85>;
pub type U86 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P86 = PInt<U86>; pub type N86 = NInt<U86>;
pub type U87 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P87 = PInt<U87>; pub type N87 = NInt<U87>;
pub type U88 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P88 = PInt<U88>; pub type N88 = NInt<U88>;
pub type U89 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P89 = PInt<U89>; pub type N89 = NInt<U89>;
pub type U90 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P90 = PInt<U90>; pub type N90 = NInt<U90>;
pub type U91 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P91 = PInt<U91>; pub type N91 = NInt<U91>;
pub type U92 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P92 = PInt<U92>; pub type N92 = NInt<U92>;
pub type U93 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P93 = PInt<U93>; pub type N93 = NInt<U93>;
pub type U94 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P94 = PInt<U94>; pub type N94 = NInt<U94>;
pub type U95 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P95 = PInt<U95>; pub type N95 = NInt<U95>;
pub type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P96 = PInt<U96>; pub type N96 = NInt<U96>;
pub type U97 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P97 = PInt<U97>; pub type N97 = NInt<U97>;
pub type U98 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
```

```
pub type P98 = PInt<U98>; pub type N98 = NInt<U98>;
pub type U99 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P99 = PInt<U99>; pub type N99 = NInt<U99>;
pub type U100 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P100 = PInt<U100>; pub type N100 = NInt<U100>;
pub type U101 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P101 = PInt<U101>; pub type N101 = NInt<U101>;
pub type U102 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P102 = PInt<U102>; pub type N102 = NInt<U102>;
pub type U103 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P103 = PInt<U103>; pub type N103 = NInt<U103>;
pub type U104 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P104 = PInt<U104>; pub type N104 = NInt<U104>;
pub type U105 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P105 = PInt<U105>; pub type N105 = NInt<U105>;
pub type U106 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P106 = PInt<U106>; pub type N106 = NInt<U106>;
pub type U107 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P107 = PInt<U107>; pub type N107 = NInt<U107>;
pub type U108 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P108 = PInt<U108>; pub type N108 = NInt<U108>;
pub type U109 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P109 = PInt<U109>; pub type N109 = NInt<U109>;
pub type U110 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P110 = PInt<U110>; pub type N110 = NInt<U110>;
pub type U111 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P111 = PInt<U111>; pub type N111 = NInt<U111>;
pub type U112 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P112 = PInt<U112>; pub type N112 = NInt<U112>;
pub type U113 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P113 = PInt<U113>; pub type N113 = NInt<U113>;
pub type U114 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P114 = PInt<U114>; pub type N114 = NInt<U114>;
pub type U115 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P115 = PInt<U115>; pub type N115 = NInt<U115>;
pub type U116 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P116 = PInt<U116>; pub type N116 = NInt<U116>;
pub type U117 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P117 = PInt<U117>; pub type N117 = NInt<U117>;
pub type U118 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P118 = PInt<U118>; pub type N118 = NInt<U118>;
pub type U119 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P119 = PInt<U119>; pub type N119 = NInt<U119>;
pub type U120 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P120 = PInt<U120>; pub type N120 = NInt<U120>;
pub type U121 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P121 = PInt<U121>; pub type N121 = NInt<U121>;
pub type U122 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P122 = PInt<U122>; pub type N122 = NInt<U122>;
pub type U123 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P123 = PInt<U123>; pub type N123 = NInt<U123>;
pub type U124 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
```

```
pub type P124 = PInt<U124>; pub type N124 = NInt<U124>;
pub type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P125 = PInt<U125>; pub type N125 = NInt<U125>;
pub type U126 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P126 = PInt<U126>; pub type N126 = NInt<U126>;
pub type U127 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P127 = PInt<U127>; pub type N127 = NInt<U127>;
pub type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P128 = PInt<U128>; pub type N128 = NInt<U128>;
pub type U129 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P129 = PInt<U129>; pub type N129 = NInt<U129>;
pub type U130 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P130 = PInt<U130>; pub type N130 = NInt<U130>;
pub type U131 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P131 = PInt<U131>; pub type N131 = NInt<U131>;
pub type U132 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P132 = PInt<U132>; pub type N132 = NInt<U132>;
pub type U133 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P133 = PInt<U133>; pub type N133 = NInt<U133>;
pub type U134 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P134 = PInt<U134>; pub type N134 = NInt<U134>;
pub type U135 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P135 = PInt<U135>; pub type N135 = NInt<U135>;
pub type U136 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P136 = PInt<U136>; pub type N136 = NInt<U136>;
pub type U137 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P137 = PInt<U137>; pub type N137 = NInt<U137>;
pub type U138 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P138 = PInt<U138>; pub type N138 = NInt<U138>;
pub type U139 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P139 = PInt<U139>; pub type N139 = NInt<U139>;
pub type U140 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P140 = PInt<U140>; pub type N140 = NInt<U140>;
pub type U141 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P141 = PInt<U141>; pub type N141 = NInt<U141>;
pub type U142 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P142 = PInt<U142>; pub type N142 = NInt<U142>;
pub type U143 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P143 = PInt<U143>; pub type N143 = NInt<U143>;
pub type U144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P144 = PInt<U144>; pub type N144 = NInt<U144>;
pub type U145 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P145 = PInt<U145>; pub type N145 = NInt<U145>;
pub type U146 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P146 = PInt<U146>; pub type N146 = NInt<U146>;
pub type U147 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P147 = PInt<U147>; pub type N147 = NInt<U147>;
pub type U148 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P148 = PInt<U148>; pub type N148 = NInt<U148>;
pub type U149 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P149 = PInt<U149>; pub type N149 = NInt<U149>;
pub type U150 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
```

```
pub type P150 = PInt<U150>; pub type N150 = NInt<U150>;
pub type U151 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P151 = PInt<U151>; pub type N151 = NInt<U151>;
pub type U152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P152 = PInt<U152>; pub type N152 = NInt<U152>;
pub type U153 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P153 = PInt<U153>; pub type N153 = NInt<U153>;
pub type U154 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P154 = PInt<U154>; pub type N154 = NInt<U154>;
pub type U155 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P155 = PInt<U155>; pub type N155 = NInt<U155>;
pub type U156 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P156 = PInt<U156>; pub type N156 = NInt<U156>;
pub type U157 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P157 = PInt<U157>; pub type N157 = NInt<U157>;
pub type U158 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P158 = PInt<U158>; pub type N158 = NInt<U158>;
pub type U159 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P159 = PInt<U159>; pub type N159 = NInt<U159>;
pub type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P160 = PInt<U160>; pub type N160 = NInt<U160>;
pub type U161 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P161 = PInt<U161>; pub type N161 = NInt<U161>;
pub type U162 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P162 = PInt<U162>; pub type N162 = NInt<U162>;
pub type U163 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P163 = PInt<U163>; pub type N163 = NInt<U163>;
pub type U164 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P164 = PInt<U164>; pub type N164 = NInt<U164>;
pub type U165 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P165 = PInt<U165>; pub type N165 = NInt<U165>;
pub type U166 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P166 = PInt<U166>; pub type N166 = NInt<U166>;
pub type U167 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P167 = PInt<U167>; pub type N167 = NInt<U167>;
pub type U168 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P168 = PInt<U168>; pub type N168 = NInt<U168>;
pub type U169 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P169 = PInt<U169>; pub type N169 = NInt<U169>;
pub type U170 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P170 = PInt<U170>; pub type N170 = NInt<U170>;
pub type U171 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P171 = PInt<U171>; pub type N171 = NInt<U171>;
pub type U172 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P172 = PInt<U172>; pub type N172 = NInt<U172>;
pub type U173 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P173 = PInt<U173>; pub type N173 = NInt<U173>;
pub type U174 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P174 = PInt<U174>; pub type N174 = NInt<U174>;
pub type U175 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P175 = PInt<U175>; pub type N175 = NInt<U175>;
pub type U176 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
```

```
pub type P176 = PInt<U176>; pub type N176 = NInt<U176>;
pub type U177 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P177 = PInt<U177>; pub type N177 = NInt<U177>;
pub type U178 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P178 = PInt<U178>; pub type N178 = NInt<U178>;
pub type U179 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P179 = PInt<U179>; pub type N179 = NInt<U179>;
pub type U180 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P180 = PInt<U180>; pub type N180 = NInt<U180>;
pub type U181 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P181 = PInt<U181>; pub type N181 = NInt<U181>;
pub type U182 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P182 = PInt<U182>; pub type N182 = NInt<U182>;
pub type U183 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P183 = PInt<U183>; pub type N183 = NInt<U183>;
pub type U184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P184 = PInt<U184>; pub type N184 = NInt<U184>;
pub type U185 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P185 = PInt<U185>; pub type N185 = NInt<U185>;
pub type U186 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P186 = PInt<U186>; pub type N186 = NInt<U186>;
pub type U187 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P187 = PInt<U187>; pub type N187 = NInt<U187>;
pub type U188 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P188 = PInt<U188>; pub type N188 = NInt<U188>;
pub type U189 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P189 = PInt<U189>; pub type N189 = NInt<U189>;
pub type U190 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P190 = PInt<U190>; pub type N190 = NInt<U190>;
pub type U191 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P191 = PInt<U191>; pub type N191 = NInt<U191>;
pub type U192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P192 = PInt<U192>; pub type N192 = NInt<U192>;
pub type U193 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P193 = PInt<U193>; pub type N193 = NInt<U193>;
pub type U194 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P194 = PInt<U194>; pub type N194 = NInt<U194>;
pub type U195 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P195 = PInt<U195>; pub type N195 = NInt<U195>;
pub type U196 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P196 = PInt<U196>; pub type N196 = NInt<U196>;
pub type U197 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P197 = PInt<U197>; pub type N197 = NInt<U197>;
pub type U198 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P198 = PInt<U198>; pub type N198 = NInt<U198>;
pub type U199 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P199 = PInt<U199>; pub type N199 = NInt<U199>;
pub type U200 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P200 = PInt<U200>; pub type N200 = NInt<U200>;
pub type U201 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P201 = PInt<U201>; pub type N201 = NInt<U201>;
pub type U202 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P202 = PInt<U202>; pub type N202 = NInt<U202>;
pub type U203 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P203 = PInt<U203>; pub type N203 = NInt<U203>;
pub type U204 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P204 = PInt<U204>; pub type N204 = NInt<U204>;
pub type U205 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P205 = PInt<U205>; pub type N205 = NInt<U205>;
pub type U206 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P206 = PInt<U206>; pub type N206 = NInt<U206>;
pub type U207 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P207 = PInt<U207>; pub type N207 = NInt<U207>;
pub type U208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P208 = PInt<U208>; pub type N208 = NInt<U208>;
pub type U209 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P209 = PInt<U209>; pub type N209 = NInt<U209>;
pub type U210 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P210 = PInt<U210>; pub type N210 = NInt<U210>;
pub type U211 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P211 = PInt<U211>; pub type N211 = NInt<U211>;
pub type U212 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P212 = PInt<U212>; pub type N212 = NInt<U212>;
pub type U213 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P213 = PInt<U213>; pub type N213 = NInt<U213>;
pub type U214 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P214 = PInt<U214>; pub type N214 = NInt<U214>;
pub type U215 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P215 = PInt<U215>; pub type N215 = NInt<U215>;
pub type U216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P216 = PInt<U216>; pub type N216 = NInt<U216>;
pub type U217 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P217 = PInt<U217>; pub type N217 = NInt<U217>;
pub type U218 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P218 = PInt<U218>; pub type N218 = NInt<U218>;
pub type U219 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P219 = PInt<U219>; pub type N219 = NInt<U219>;
pub type U220 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P220 = PInt<U220>; pub type N220 = NInt<U220>;
pub type U221 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P221 = PInt<U221>; pub type N221 = NInt<U221>;
pub type U222 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P222 = PInt<U222>; pub type N222 = NInt<U222>;
pub type U223 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P223 = PInt<U223>; pub type N223 = NInt<U223>;
pub type U224 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P224 = PInt<U224>; pub type N224 = NInt<U224>;
pub type U225 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P225 = PInt<U225>; pub type N225 = NInt<U225>;
pub type U226 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P226 = PInt<U226>; pub type N226 = NInt<U226>;
pub type U227 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P227 = PInt<U227>; pub type N227 = NInt<U227>;
pub type U228 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P228 = PInt<U228>; pub type N228 = NInt<U228>;
pub type U229 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P229 = PInt<U229>; pub type N229 = NInt<U229>;
pub type U230 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P230 = PInt<U230>; pub type N230 = NInt<U230>;
pub type U231 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P231 = PInt<U231>; pub type N231 = NInt<U231>;
pub type U232 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P232 = PInt<U232>; pub type N232 = NInt<U232>;
pub type U233 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P233 = PInt<U233>; pub type N233 = NInt<U233>;
pub type U234 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P234 = PInt<U234>; pub type N234 = NInt<U234>;
pub type U235 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P235 = PInt<U235>; pub type N235 = NInt<U235>;
pub type U236 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P236 = PInt<U236>; pub type N236 = NInt<U236>;
pub type U237 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P237 = PInt<U237>; pub type N237 = NInt<U237>;
pub type U238 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P238 = PInt<U238>; pub type N238 = NInt<U238>;
pub type U239 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P239 = PInt<U239>; pub type N239 = NInt<U239>;
pub type U240 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P240 = PInt<U240>; pub type N240 = NInt<U240>;
pub type U241 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P241 = PInt<U241>; pub type N241 = NInt<U241>;
pub type U242 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P242 = PInt<U242>; pub type N242 = NInt<U242>;
pub type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P243 = PInt<U243>; pub type N243 = NInt<U243>;
pub type U244 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P244 = PInt<U244>; pub type N244 = NInt<U244>;
pub type U245 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P245 = PInt<U245>; pub type N245 = NInt<U245>;
pub type U246 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P246 = PInt<U246>; pub type N246 = NInt<U246>;
pub type U247 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P247 = PInt<U247>; pub type N247 = NInt<U247>;
pub type U248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P248 = PInt<U248>; pub type N248 = NInt<U248>;
pub type U249 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P249 = PInt<U249>; pub type N249 = NInt<U249>;
pub type U250 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P250 = PInt<U250>; pub type N250 = NInt<U250>;
pub type U251 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P251 = PInt<U251>; pub type N251 = NInt<U251>;
pub type U252 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P252 = PInt<U252>; pub type N252 = NInt<U252>;
pub type U253 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P253 = PInt<U253>; pub type N253 = NInt<U253>;
pub type U254 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P254 = PInt<U254>; pub type N254 = NInt<U254>;
pub type U255 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P255 = PInt<U255>; pub type N255 = NInt<U255>;
pub type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P256 = PInt<U256>; pub type N256 = NInt<U256>;
pub type U257 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P257 = PInt<U257>; pub type N257 = NInt<U257>;
pub type U258 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P258 = PInt<U258>; pub type N258 = NInt<U258>;
pub type U259 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P259 = PInt<U259>; pub type N259 = NInt<U259>;
pub type U260 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P260 = PInt<U260>; pub type N260 = NInt<U260>;
pub type U261 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P261 = PInt<U261>; pub type N261 = NInt<U261>;
pub type U262 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P262 = PInt<U262>; pub type N262 = NInt<U262>;
pub type U263 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P263 = PInt<U263>; pub type N263 = NInt<U263>;
pub type U264 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P264 = PInt<U264>; pub type N264 = NInt<U264>;
pub type U265 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P265 = PInt<U265>; pub type N265 = NInt<U265>;
pub type U266 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P266 = PInt<U266>; pub type N266 = NInt<U266>;
pub type U267 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P267 = PInt<U267>; pub type N267 = NInt<U267>;
pub type U268 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P268 = PInt<U268>; pub type N268 = NInt<U268>;
pub type U269 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P269 = PInt<U269>; pub type N269 = NInt<U269>;
pub type U270 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P270 = PInt<U270>; pub type N270 = NInt<U270>;
pub type U271 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P271 = PInt<U271>; pub type N271 = NInt<U271>;
pub type U272 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P272 = PInt<U272>; pub type N272 = NInt<U272>;
pub type U273 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P273 = PInt<U273>; pub type N273 = NInt<U273>;
pub type U274 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P274 = PInt<U274>; pub type N274 = NInt<U274>;
pub type U275 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P275 = PInt<U275>; pub type N275 = NInt<U275>;
pub type U276 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P276 = PInt<U276>; pub type N276 = NInt<U276>;
pub type U277 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P277 = PInt<U277>; pub type N277 = NInt<U277>;
pub type U278 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P278 = PInt<U278>; pub type N278 = NInt<U278>;
pub type U279 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P279 = PInt<U279>; pub type N279 = NInt<U279>;
pub type U280 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P280 = PInt<U280>; pub type N280 = NInt<U280>;
pub type U281 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P281 = PInt<U281>; pub type N281 = NInt<U281>;
pub type U282 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P282 = PInt<U282>; pub type N282 = NInt<U282>;
pub type U283 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P283 = PInt<U283>; pub type N283 = NInt<U283>;
pub type U284 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P284 = PInt<U284>; pub type N284 = NInt<U284>;
pub type U285 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P285 = PInt<U285>; pub type N285 = NInt<U285>;
pub type U286 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P286 = PInt<U286>; pub type N286 = NInt<U286>;
pub type U287 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P287 = PInt<U287>; pub type N287 = NInt<U287>;
pub type U288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P288 = PInt<U288>; pub type N288 = NInt<U288>;
pub type U289 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P289 = PInt<U289>; pub type N289 = NInt<U289>;
pub type U290 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P290 = PInt<U290>; pub type N290 = NInt<U290>;
pub type U291 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P291 = PInt<U291>; pub type N291 = NInt<U291>;
pub type U292 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P292 = PInt<U292>; pub type N292 = NInt<U292>;
pub type U293 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P293 = PInt<U293>; pub type N293 = NInt<U293>;
pub type U294 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P294 = PInt<U294>; pub type N294 = NInt<U294>;
pub type U295 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P295 = PInt<U295>; pub type N295 = NInt<U295>;
pub type U296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P296 = PInt<U296>; pub type N296 = NInt<U296>;
pub type U297 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P297 = PInt<U297>; pub type N297 = NInt<U297>;
pub type U298 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P298 = PInt<U298>; pub type N298 = NInt<U298>;
pub type U299 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P299 = PInt<U299>; pub type N299 = NInt<U299>;
pub type U300 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P300 = PInt<U300>; pub type N300 = NInt<U300>;
pub type U301 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P301 = PInt<U301>; pub type N301 = NInt<U301>;
pub type U302 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P302 = PInt<U302>; pub type N302 = NInt<U302>;
pub type U303 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P303 = PInt<U303>; pub type N303 = NInt<U303>;
pub type U304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P304 = PInt<U304>; pub type N304 = NInt<U304>;
pub type U305 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P305 = PInt<U305>; pub type N305 = NInt<U305>;
pub type U306 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P306 = PInt<U306>; pub type N306 = NInt<U306>;
pub type U307 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P307 = PInt<U307>; pub type N307 = NInt<U307>;
pub type U308 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P308 = PInt<U308>; pub type N308 = NInt<U308>;
pub type U309 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P309 = PInt<U309>; pub type N309 = NInt<U309>;
pub type U310 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P310 = PInt<U310>; pub type N310 = NInt<U310>;
pub type U311 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P311 = PInt<U311>; pub type N311 = NInt<U311>;
pub type U312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P312 = PInt<U312>; pub type N312 = NInt<U312>;
pub type U313 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P313 = PInt<U313>; pub type N313 = NInt<U313>;
pub type U314 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P314 = PInt<U314>; pub type N314 = NInt<U314>;
pub type U315 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P315 = PInt<U315>; pub type N315 = NInt<U315>;
pub type U316 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P316 = PInt<U316>; pub type N316 = NInt<U316>;
pub type U317 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P317 = PInt<U317>; pub type N317 = NInt<U317>;
pub type U318 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P318 = PInt<U318>; pub type N318 = NInt<U318>;
pub type U319 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P319 = PInt<U319>; pub type N319 = NInt<U319>;
pub type U320 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P320 = PInt<U320>; pub type N320 = NInt<U320>;
pub type U321 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P321 = PInt<U321>; pub type N321 = NInt<U321>;
pub type U322 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P322 = PInt<U322>; pub type N322 = NInt<U322>;
pub type U323 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P323 = PInt<U323>; pub type N323 = NInt<U323>;
pub type U324 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P324 = PInt<U324>; pub type N324 = NInt<U324>;
pub type U325 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P325 = PInt<U325>; pub type N325 = NInt<U325>;
pub type U326 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P326 = PInt<U326>; pub type N326 = NInt<U326>;
pub type U327 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P327 = PInt<U327>; pub type N327 = NInt<U327>;
pub type U328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P328 = PInt<U328>; pub type N328 = NInt<U328>;
pub type U329 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P329 = PInt<U329>; pub type N329 = NInt<U329>;
pub type U330 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P330 = PInt<U330>; pub type N330 = NInt<U330>;
pub type U331 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P331 = PInt<U331>; pub type N331 = NInt<U331>;
pub type U332 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P332 = PInt<U332>; pub type N332 = NInt<U332>;
pub type U333 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P333 = PInt<U333>; pub type N333 = NInt<U333>;
pub type U334 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P334 = PInt<U334>; pub type N334 = NInt<U334>;
pub type U335 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P335 = PInt<U335>; pub type N335 = NInt<U335>;
pub type U336 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P336 = PInt<U336>; pub type N336 = NInt<U336>;
pub type U337 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P337 = PInt<U337>; pub type N337 = NInt<U337>;
pub type U338 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P338 = PInt<U338>; pub type N338 = NInt<U338>;
pub type U339 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P339 = PInt<U339>; pub type N339 = NInt<U339>;
pub type U340 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P340 = PInt<U340>; pub type N340 = NInt<U340>;
pub type U341 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P341 = PInt<U341>; pub type N341 = NInt<U341>;
pub type U342 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P342 = PInt<U342>; pub type N342 = NInt<U342>;
pub type U343 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P343 = PInt<U343>; pub type N343 = NInt<U343>;
pub type U344 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P344 = PInt<U344>; pub type N344 = NInt<U344>;
pub type U345 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P345 = PInt<U345>; pub type N345 = NInt<U345>;
pub type U346 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P346 = PInt<U346>; pub type N346 = NInt<U346>;
pub type U347 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P347 = PInt<U347>; pub type N347 = NInt<U347>;
pub type U348 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P348 = PInt<U348>; pub type N348 = NInt<U348>;
pub type U349 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P349 = PInt<U349>; pub type N349 = NInt<U349>;
pub type U350 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P350 = PInt<U350>; pub type N350 = NInt<U350>;
pub type U351 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P351 = PInt<U351>; pub type N351 = NInt<U351>;
pub type U352 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P352 = PInt<U352>; pub type N352 = NInt<U352>;
pub type U353 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P353 = PInt<U353>; pub type N353 = NInt<U353>;
pub type U354 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P354 = PInt<U354>; pub type N354 = NInt<U354>;
pub type U355 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P355 = PInt<U355>; pub type N355 = NInt<U355>;
pub type U356 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P356 = PInt<U356>; pub type N356 = NInt<U356>;
pub type U357 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P357 = PInt<U357>; pub type N357 = NInt<U357>;
pub type U358 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P358 = PInt<U358>; pub type N358 = NInt<U358>;
pub type U359 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P359 = PInt<U359>; pub type N359 = NInt<U359>;
pub type U360 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P360 = PInt<U360>; pub type N360 = NInt<U360>;
pub type U361 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P361 = PInt<U361>; pub type N361 = NInt<U361>;
pub type U362 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P362 = PInt<U362>; pub type N362 = NInt<U362>;
pub type U363 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P363 = PInt<U363>; pub type N363 = NInt<U363>;
pub type U364 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P364 = PInt<U364>; pub type N364 = NInt<U364>;
pub type U365 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P365 = PInt<U365>; pub type N365 = NInt<U365>;
pub type U366 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P366 = PInt<U366>; pub type N366 = NInt<U366>;
pub type U367 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P367 = PInt<U367>; pub type N367 = NInt<U367>;
pub type U368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P368 = PInt<U368>; pub type N368 = NInt<U368>;
pub type U369 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P369 = PInt<U369>; pub type N369 = NInt<U369>;
pub type U370 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P370 = PInt<U370>; pub type N370 = NInt<U370>;
pub type U371 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P371 = PInt<U371>; pub type N371 = NInt<U371>;
pub type U372 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P372 = PInt<U372>; pub type N372 = NInt<U372>;
pub type U373 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P373 = PInt<U373>; pub type N373 = NInt<U373>;
pub type U374 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P374 = PInt<U374>; pub type N374 = NInt<U374>;
pub type U375 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P375 = PInt<U375>; pub type N375 = NInt<U375>;
pub type U376 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P376 = PInt<U376>; pub type N376 = NInt<U376>;
pub type U377 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P377 = PInt<U377>; pub type N377 = NInt<U377>;
pub type U378 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P378 = PInt<U378>; pub type N378 = NInt<U378>;
pub type U379 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P379 = PInt<U379>; pub type N379 = NInt<U379>;
pub type U380 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P380 = PInt<U380>; pub type N380 = NInt<U380>;
pub type U381 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P381 = PInt<U381>; pub type N381 = NInt<U381>;
pub type U382 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P382 = PInt<U382>; pub type N382 = NInt<U382>;
pub type U383 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P383 = PInt<U383>; pub type N383 = NInt<U383>;
pub type U384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P384 = PInt<U384>; pub type N384 = NInt<U384>;
pub type U385 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P385 = PInt<U385>; pub type N385 = NInt<U385>;
pub type U386 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P386 = PInt<U386>; pub type N386 = NInt<U386>;
pub type U387 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P387 = PInt<U387>; pub type N387 = NInt<U387>;
pub type U388 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P388 = PInt<U388>; pub type N388 = NInt<U388>;
pub type U389 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P389 = PInt<U389>; pub type N389 = NInt<U389>;
pub type U390 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P390 = PInt<U390>; pub type N390 = NInt<U390>;
pub type U391 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P391 = PInt<U391>; pub type N391 = NInt<U391>;
pub type U392 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P392 = PInt<U392>; pub type N392 = NInt<U392>;
pub type U393 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P393 = PInt<U393>; pub type N393 = NInt<U393>;
pub type U394 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P394 = PInt<U394>; pub type N394 = NInt<U394>;
pub type U395 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P395 = PInt<U395>; pub type N395 = NInt<U395>;
pub type U396 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P396 = PInt<U396>; pub type N396 = NInt<U396>;
pub type U397 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P397 = PInt<U397>; pub type N397 = NInt<U397>;
pub type U398 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P398 = PInt<U398>; pub type N398 = NInt<U398>;
pub type U399 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P399 = PInt<U399>; pub type N399 = NInt<U399>;
pub type U400 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P400 = PInt<U400>; pub type N400 = NInt<U400>;
pub type U401 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P401 = PInt<U401>; pub type N401 = NInt<U401>;
pub type U402 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P402 = PInt<U402>; pub type N402 = NInt<U402>;
pub type U403 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P403 = PInt<U403>; pub type N403 = NInt<U403>;
pub type U404 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P404 = PInt<U404>; pub type N404 = NInt<U404>;
pub type U405 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P405 = PInt<U405>; pub type N405 = NInt<U405>;
pub type U406 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P406 = PInt<U406>; pub type N406 = NInt<U406>;
pub type U407 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P407 = PInt<U407>; pub type N407 = NInt<U407>;
pub type U408 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P408 = PInt<U408>; pub type N408 = NInt<U408>;
pub type U409 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P409 = PInt<U409>; pub type N409 = NInt<U409>;
pub type U410 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P410 = PInt<U410>; pub type N410 = NInt<U410>;
pub type U411 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P411 = PInt<U411>; pub type N411 = NInt<U411>;
pub type U412 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P412 = PInt<U412>; pub type N412 = NInt<U412>;
pub type U413 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P413 = PInt<U413>; pub type N413 = NInt<U413>;
pub type U414 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P414 = PInt<U414>; pub type N414 = NInt<U414>;
pub type U415 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P415 = PInt<U415>; pub type N415 = NInt<U415>;
pub type U416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P416 = PInt<U416>; pub type N416 = NInt<U416>;
pub type U417 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P417 = PInt<U417>; pub type N417 = NInt<U417>;
pub type U418 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P418 = PInt<U418>; pub type N418 = NInt<U418>;
pub type U419 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P419 = PInt<U419>; pub type N419 = NInt<U419>;
pub type U420 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P420 = PInt<U420>; pub type N420 = NInt<U420>;
pub type U421 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P421 = PInt<U421>; pub type N421 = NInt<U421>;
pub type U422 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P422 = PInt<U422>; pub type N422 = NInt<U422>;
pub type U423 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P423 = PInt<U423>; pub type N423 = NInt<U423>;
pub type U424 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P424 = PInt<U424>; pub type N424 = NInt<U424>;
pub type U425 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P425 = PInt<U425>; pub type N425 = NInt<U425>;
pub type U426 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P426 = PInt<U426>; pub type N426 = NInt<U426>;
pub type U427 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P427 = PInt<U427>; pub type N427 = NInt<U427>;
pub type U428 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P428 = PInt<U428>; pub type N428 = NInt<U428>;
pub type U429 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P429 = PInt<U429>; pub type N429 = NInt<U429>;
pub type U430 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P430 = PInt<U430>; pub type N430 = NInt<U430>;
pub type U431 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P431 = PInt<U431>; pub type N431 = NInt<U431>;
pub type U432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P432 = PInt<U432>; pub type N432 = NInt<U432>;
pub type U433 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P433 = PInt<U433>; pub type N433 = NInt<U433>;
pub type U434 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P434 = PInt<U434>; pub type N434 = NInt<U434>;
pub type U435 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P435 = PInt<U435>; pub type N435 = NInt<U435>;
pub type U436 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P436 = PInt<U436>; pub type N436 = NInt<U436>;
pub type U437 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P437 = PInt<U437>; pub type N437 = NInt<U437>;
pub type U438 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P438 = PInt<U438>; pub type N438 = NInt<U438>;
pub type U439 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P439 = PInt<U439>; pub type N439 = NInt<U439>;
pub type U440 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P440 = PInt<U440>; pub type N440 = NInt<U440>;
pub type U441 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P441 = PInt<U441>; pub type N441 = NInt<U441>;
pub type U442 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P442 = PInt<U442>; pub type N442 = NInt<U442>;
pub type U443 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P443 = PInt<U443>; pub type N443 = NInt<U443>;
pub type U444 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P444 = PInt<U444>; pub type N444 = NInt<U444>;
pub type U445 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P445 = PInt<U445>; pub type N445 = NInt<U445>;
pub type U446 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P446 = PInt<U446>; pub type N446 = NInt<U446>;
pub type U447 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P447 = PInt<U447>; pub type N447 = NInt<U447>;
pub type U448 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P448 = PInt<U448>; pub type N448 = NInt<U448>;
pub type U449 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P449 = PInt<U449>; pub type N449 = NInt<U449>;
pub type U450 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P450 = PInt<U450>; pub type N450 = NInt<U450>;
pub type U451 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P451 = PInt<U451>; pub type N451 = NInt<U451>;
pub type U452 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P452 = PInt<U452>; pub type N452 = NInt<U452>;
pub type U453 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P453 = PInt<U453>; pub type N453 = NInt<U453>;
pub type U454 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P454 = PInt<U454>; pub type N454 = NInt<U454>;
pub type U455 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P455 = PInt<U455>; pub type N455 = NInt<U455>;
pub type U456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P456 = PInt<U456>; pub type N456 = NInt<U456>;
pub type U457 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P457 = PInt<U457>; pub type N457 = NInt<U457>;
pub type U458 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P458 = PInt<U458>; pub type N458 = NInt<U458>;
pub type U459 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P459 = PInt<U459>; pub type N459 = NInt<U459>;
pub type U460 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P460 = PInt<U460>; pub type N460 = NInt<U460>;
pub type U461 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P461 = PInt<U461>; pub type N461 = NInt<U461>;
pub type U462 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P462 = PInt<U462>; pub type N462 = NInt<U462>;
pub type U463 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P463 = PInt<U463>; pub type N463 = NInt<U463>;
pub type U464 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P464 = PInt<U464>; pub type N464 = NInt<U464>;
pub type U465 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P465 = PInt<U465>; pub type N465 = NInt<U465>;
pub type U466 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P466 = PInt<U466>; pub type N466 = NInt<U466>;
pub type U467 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P467 = PInt<U467>; pub type N467 = NInt<U467>;
pub type U468 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P468 = PInt<U468>; pub type N468 = NInt<U468>;
pub type U469 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P469 = PInt<U469>; pub type N469 = NInt<U469>;
pub type U470 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P470 = PInt<U470>; pub type N470 = NInt<U470>;
pub type U471 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P471 = PInt<U471>; pub type N471 = NInt<U471>;
pub type U472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P472 = PInt<U472>; pub type N472 = NInt<U472>;
pub type U473 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P473 = PInt<U473>; pub type N473 = NInt<U473>;
pub type U474 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P474 = PInt<U474>; pub type N474 = NInt<U474>;
pub type U475 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P475 = PInt<U475>; pub type N475 = NInt<U475>;
pub type U476 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P476 = PInt<U476>; pub type N476 = NInt<U476>;
pub type U477 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P477 = PInt<U477>; pub type N477 = NInt<U477>;
pub type U478 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P478 = PInt<U478>; pub type N478 = NInt<U478>;
pub type U479 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P479 = PInt<U479>; pub type N479 = NInt<U479>;
pub type U480 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P480 = PInt<U480>; pub type N480 = NInt<U480>;
pub type U481 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P481 = PInt<U481>; pub type N481 = NInt<U481>;
pub type U482 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P482 = PInt<U482>; pub type N482 = NInt<U482>;
pub type U483 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P483 = PInt<U483>; pub type N483 = NInt<U483>;
pub type U484 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P484 = PInt<U484>; pub type N484 = NInt<U484>;
pub type U485 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P485 = PInt<U485>; pub type N485 = NInt<U485>;
pub type U486 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P486 = PInt<U486>; pub type N486 = NInt<U486>;
pub type U487 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P487 = PInt<U487>; pub type N487 = NInt<U487>;
pub type U488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P488 = PInt<U488>; pub type N488 = NInt<U488>;
pub type U489 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P489 = PInt<U489>; pub type N489 = NInt<U489>;
pub type U490 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P490 = PInt<U490>; pub type N490 = NInt<U490>;
pub type U491 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P491 = PInt<U491>; pub type N491 = NInt<U491>;
pub type U492 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P492 = PInt<U492>; pub type N492 = NInt<U492>;
pub type U493 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P493 = PInt<U493>; pub type N493 = NInt<U493>;
pub type U494 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P494 = PInt<U494>; pub type N494 = NInt<U494>;
pub type U495 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P495 = PInt<U495>; pub type N495 = NInt<U495>;
pub type U496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P496 = PInt<U496>; pub type N496 = NInt<U496>;
pub type U497 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P497 = PInt<U497>; pub type N497 = NInt<U497>;
pub type U498 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P498 = PInt<U498>; pub type N498 = NInt<U498>;
pub type U499 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P499 = PInt<U499>; pub type N499 = NInt<U499>;
pub type U500 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P500 = PInt<U500>; pub type N500 = NInt<U500>;
pub type U501 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P501 = PInt<U501>; pub type N501 = NInt<U501>;
pub type U502 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P502 = PInt<U502>; pub type N502 = NInt<U502>;
pub type U503 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P503 = PInt<U503>; pub type N503 = NInt<U503>;
pub type U504 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P504 = PInt<U504>; pub type N504 = NInt<U504>;
pub type U505 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P505 = PInt<U505>; pub type N505 = NInt<U505>;
pub type U506 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P506 = PInt<U506>; pub type N506 = NInt<U506>;
pub type U507 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P507 = PInt<U507>; pub type N507 = NInt<U507>;
pub type U508 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P508 = PInt<U508>; pub type N508 = NInt<U508>;
pub type U509 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P509 = PInt<U509>; pub type N509 = NInt<U509>;
pub type U510 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P510 = PInt<U510>; pub type N510 = NInt<U510>;
pub type U511 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P511 = PInt<U511>; pub type N511 = NInt<U511>;
pub type U512 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P512 = PInt<U512>; pub type N512 = NInt<U512>;
pub type U513 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P513 = PInt<U513>; pub type N513 = NInt<U513>;
pub type U514 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P514 = PInt<U514>; pub type N514 = NInt<U514>;
pub type U515 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P515 = PInt<U515>; pub type N515 = NInt<U515>;
pub type U516 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P516 = PInt<U516>; pub type N516 = NInt<U516>;
pub type U517 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P517 = PInt<U517>; pub type N517 = NInt<U517>;
pub type U518 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P518 = PInt<U518>; pub type N518 = NInt<U518>;
pub type U519 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P519 = PInt<U519>; pub type N519 = NInt<U519>;
pub type U520 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P520 = PInt<U520>; pub type N520 = NInt<U520>;
pub type U521 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P521 = PInt<U521>; pub type N521 = NInt<U521>;
pub type U522 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P522 = PInt<U522>; pub type N522 = NInt<U522>;
pub type U523 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P523 = PInt<U523>; pub type N523 = NInt<U523>;
pub type U524 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P524 = PInt<U524>; pub type N524 = NInt<U524>;
pub type U525 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P525 = PInt<U525>; pub type N525 = NInt<U525>;
pub type U526 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P526 = PInt<U526>; pub type N526 = NInt<U526>;
pub type U527 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P527 = PInt<U527>; pub type N527 = NInt<U527>;
pub type U528 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P528 = PInt<U528>; pub type N528 = NInt<U528>;
pub type U529 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P529 = PInt<U529>; pub type N529 = NInt<U529>;
pub type U530 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P530 = PInt<U530>; pub type N530 = NInt<U530>;
pub type U531 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P531 = PInt<U531>; pub type N531 = NInt<U531>;
pub type U532 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P532 = PInt<U532>; pub type N532 = NInt<U532>;
pub type U533 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P533 = PInt<U533>; pub type N533 = NInt<U533>;
pub type U534 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P534 = PInt<U534>; pub type N534 = NInt<U534>;
pub type U535 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P535 = PInt<U535>; pub type N535 = NInt<U535>;
pub type U536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P536 = PInt<U536>; pub type N536 = NInt<U536>;
pub type U537 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P537 = PInt<U537>; pub type N537 = NInt<U537>;
pub type U538 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P538 = PInt<U538>; pub type N538 = NInt<U538>;
pub type U539 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P539 = PInt<U539>; pub type N539 = NInt<U539>;
pub type U540 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P540 = PInt<U540>; pub type N540 = NInt<U540>;
pub type U541 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P541 = PInt<U541>; pub type N541 = NInt<U541>;
pub type U542 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P542 = PInt<U542>; pub type N542 = NInt<U542>;
pub type U543 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P543 = PInt<U543>; pub type N543 = NInt<U543>;
pub type U544 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P544 = PInt<U544>; pub type N544 = NInt<U544>;
pub type U545 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P545 = PInt<U545>; pub type N545 = NInt<U545>;
pub type U546 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P546 = PInt<U546>; pub type N546 = NInt<U546>;
pub type U547 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P547 = PInt<U547>; pub type N547 = NInt<U547>;
pub type U548 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P548 = PInt<U548>; pub type N548 = NInt<U548>;
pub type U549 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P549 = PInt<U549>; pub type N549 = NInt<U549>;
pub type U550 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P550 = PInt<U550>; pub type N550 = NInt<U550>;
pub type U551 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P551 = PInt<U551>; pub type N551 = NInt<U551>;
pub type U552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P552 = PInt<U552>; pub type N552 = NInt<U552>;
pub type U553 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P553 = PInt<U553>; pub type N553 = NInt<U553>;
pub type U554 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P554 = PInt<U554>; pub type N554 = NInt<U554>;
pub type U555 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P555 = PInt<U555>; pub type N555 = NInt<U555>;
pub type U556 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P556 = PInt<U556>; pub type N556 = NInt<U556>;
pub type U557 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P557 = PInt<U557>; pub type N557 = NInt<U557>;
pub type U558 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P558 = PInt<U558>; pub type N558 = NInt<U558>;
pub type U559 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P559 = PInt<U559>; pub type N559 = NInt<U559>;
pub type U560 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P560 = PInt<U560>; pub type N560 = NInt<U560>;
pub type U561 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P561 = PInt<U561>; pub type N561 = NInt<U561>;
pub type U562 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P562 = PInt<U562>; pub type N562 = NInt<U562>;
pub type U563 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P563 = PInt<U563>; pub type N563 = NInt<U563>;
pub type U564 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P564 = PInt<U564>; pub type N564 = NInt<U564>;
pub type U565 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P565 = PInt<U565>; pub type N565 = NInt<U565>;
pub type U566 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P566 = PInt<U566>; pub type N566 = NInt<U566>;
pub type U567 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P567 = PInt<U567>; pub type N567 = NInt<U567>;
pub type U568 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P568 = PInt<U568>; pub type N568 = NInt<U568>;
pub type U569 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P569 = PInt<U569>; pub type N569 = NInt<U569>;
pub type U570 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P570 = PInt<U570>; pub type N570 = NInt<U570>;
pub type U571 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P571 = PInt<U571>; pub type N571 = NInt<U571>;
pub type U572 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P572 = PInt<U572>; pub type N572 = NInt<U572>;
pub type U573 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P573 = PInt<U573>; pub type N573 = NInt<U573>;
pub type U574 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P574 = PInt<U574>; pub type N574 = NInt<U574>;
pub type U575 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P575 = PInt<U575>; pub type N575 = NInt<U575>;
pub type U576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P576 = PInt<U576>; pub type N576 = NInt<U576>;
pub type U577 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P577 = PInt<U577>; pub type N577 = NInt<U577>;
pub type U578 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P578 = PInt<U578>; pub type N578 = NInt<U578>;
pub type U579 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P579 = PInt<U579>; pub type N579 = NInt<U579>;
pub type U580 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P580 = PInt<U580>; pub type N580 = NInt<U580>;
pub type U581 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P581 = PInt<U581>; pub type N581 = NInt<U581>;
pub type U582 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P582 = PInt<U582>; pub type N582 = NInt<U582>;
pub type U583 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P583 = PInt<U583>; pub type N583 = NInt<U583>;
pub type U584 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P584 = PInt<U584>; pub type N584 = NInt<U584>;
pub type U585 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P585 = PInt<U585>; pub type N585 = NInt<U585>;
pub type U586 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P586 = PInt<U586>; pub type N586 = NInt<U586>;
pub type U587 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P587 = PInt<U587>; pub type N587 = NInt<U587>;
pub type U588 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P588 = PInt<U588>; pub type N588 = NInt<U588>;
pub type U589 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P589 = PInt<U589>; pub type N589 = NInt<U589>;
pub type U590 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P590 = PInt<U590>; pub type N590 = NInt<U590>;
pub type U591 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P591 = PInt<U591>; pub type N591 = NInt<U591>;
pub type U592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P592 = PInt<U592>; pub type N592 = NInt<U592>;
pub type U593 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P593 = PInt<U593>; pub type N593 = NInt<U593>;
pub type U594 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P594 = PInt<U594>; pub type N594 = NInt<U594>;
pub type U595 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P595 = PInt<U595>; pub type N595 = NInt<U595>;
pub type U596 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P596 = PInt<U596>; pub type N596 = NInt<U596>;
pub type U597 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P597 = PInt<U597>; pub type N597 = NInt<U597>;
pub type U598 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P598 = PInt<U598>; pub type N598 = NInt<U598>;
pub type U599 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P599 = PInt<U599>; pub type N599 = NInt<U599>;
pub type U600 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P600 = PInt<U600>; pub type N600 = NInt<U600>;
pub type U601 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P601 = PInt<U601>; pub type N601 = NInt<U601>;
pub type U602 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P602 = PInt<U602>; pub type N602 = NInt<U602>;
pub type U603 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P603 = PInt<U603>; pub type N603 = NInt<U603>;
pub type U604 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P604 = PInt<U604>; pub type N604 = NInt<U604>;
pub type U605 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P605 = PInt<U605>; pub type N605 = NInt<U605>;
pub type U606 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P606 = PInt<U606>; pub type N606 = NInt<U606>;
pub type U607 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P607 = PInt<U607>; pub type N607 = NInt<U607>;
pub type U608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P608 = PInt<U608>; pub type N608 = NInt<U608>;
pub type U609 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P609 = PInt<U609>; pub type N609 = NInt<U609>;
pub type U610 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P610 = PInt<U610>; pub type N610 = NInt<U610>;
pub type U611 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P611 = PInt<U611>; pub type N611 = NInt<U611>;
pub type U612 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P612 = PInt<U612>; pub type N612 = NInt<U612>;
pub type U613 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P613 = PInt<U613>; pub type N613 = NInt<U613>;
pub type U614 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P614 = PInt<U614>; pub type N614 = NInt<U614>;
pub type U615 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P615 = PInt<U615>; pub type N615 = NInt<U615>;
pub type U616 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P616 = PInt<U616>; pub type N616 = NInt<U616>;
pub type U617 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P617 = PInt<U617>; pub type N617 = NInt<U617>;
pub type U618 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P618 = PInt<U618>; pub type N618 = NInt<U618>;
pub type U619 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P619 = PInt<U619>; pub type N619 = NInt<U619>;
pub type U620 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P620 = PInt<U620>; pub type N620 = NInt<U620>;
pub type U621 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P621 = PInt<U621>; pub type N621 = NInt<U621>;
pub type U622 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P622 = PInt<U622>; pub type N622 = NInt<U622>;
pub type U623 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P623 = PInt<U623>; pub type N623 = NInt<U623>;
pub type U624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P624 = PInt<U624>; pub type N624 = NInt<U624>;
pub type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P625 = PInt<U625>; pub type N625 = NInt<U625>;
pub type U626 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P626 = PInt<U626>; pub type N626 = NInt<U626>;
pub type U627 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P627 = PInt<U627>; pub type N627 = NInt<U627>;
pub type U628 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P628 = PInt<U628>; pub type N628 = NInt<U628>;
pub type U629 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P629 = PInt<U629>; pub type N629 = NInt<U629>;
pub type U630 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P630 = PInt<U630>; pub type N630 = NInt<U630>;
pub type U631 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P631 = PInt<U631>; pub type N631 = NInt<U631>;
pub type U632 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P632 = PInt<U632>; pub type N632 = NInt<U632>;
pub type U633 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P633 = PInt<U633>; pub type N633 = NInt<U633>;
pub type U634 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P634 = PInt<U634>; pub type N634 = NInt<U634>;
pub type U635 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P635 = PInt<U635>; pub type N635 = NInt<U635>;
pub type U636 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P636 = PInt<U636>; pub type N636 = NInt<U636>;
pub type U637 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P637 = PInt<U637>; pub type N637 = NInt<U637>;
pub type U638 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P638 = PInt<U638>; pub type N638 = NInt<U638>;
pub type U639 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P639 = PInt<U639>; pub type N639 = NInt<U639>;
pub type U640 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P640 = PInt<U640>; pub type N640 = NInt<U640>;
pub type U641 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P641 = PInt<U641>; pub type N641 = NInt<U641>;
pub type U642 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P642 = PInt<U642>; pub type N642 = NInt<U642>;
pub type U643 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P643 = PInt<U643>; pub type N643 = NInt<U643>;
pub type U644 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P644 = PInt<U644>; pub type N644 = NInt<U644>;
pub type U645 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P645 = PInt<U645>; pub type N645 = NInt<U645>;
pub type U646 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P646 = PInt<U646>; pub type N646 = NInt<U646>;
pub type U647 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P647 = PInt<U647>; pub type N647 = NInt<U647>;
pub type U648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P648 = PInt<U648>; pub type N648 = NInt<U648>;
pub type U649 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P649 = PInt<U649>; pub type N649 = NInt<U649>;
pub type U650 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P650 = PInt<U650>; pub type N650 = NInt<U650>;
pub type U651 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P651 = PInt<U651>; pub type N651 = NInt<U651>;
pub type U652 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P652 = PInt<U652>; pub type N652 = NInt<U652>;
pub type U653 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P653 = PInt<U653>; pub type N653 = NInt<U653>;
pub type U654 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P654 = PInt<U654>; pub type N654 = NInt<U654>;
pub type U655 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P655 = PInt<U655>; pub type N655 = NInt<U655>;
pub type U656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P656 = PInt<U656>; pub type N656 = NInt<U656>;
pub type U657 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P657 = PInt<U657>; pub type N657 = NInt<U657>;
pub type U658 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P658 = PInt<U658>; pub type N658 = NInt<U658>;
pub type U659 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P659 = PInt<U659>; pub type N659 = NInt<U659>;
pub type U660 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P660 = PInt<U660>; pub type N660 = NInt<U660>;
pub type U661 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P661 = PInt<U661>; pub type N661 = NInt<U661>;
pub type U662 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P662 = PInt<U662>; pub type N662 = NInt<U662>;
pub type U663 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P663 = PInt<U663>; pub type N663 = NInt<U663>;
pub type U664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P664 = PInt<U664>; pub type N664 = NInt<U664>;
pub type U665 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P665 = PInt<U665>; pub type N665 = NInt<U665>;
pub type U666 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P666 = PInt<U666>; pub type N666 = NInt<U666>;
pub type U667 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P667 = PInt<U667>; pub type N667 = NInt<U667>;
pub type U668 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P668 = PInt<U668>; pub type N668 = NInt<U668>;
pub type U669 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P669 = PInt<U669>; pub type N669 = NInt<U669>;
pub type U670 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P670 = PInt<U670>; pub type N670 = NInt<U670>;
pub type U671 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P671 = PInt<U671>; pub type N671 = NInt<U671>;
pub type U672 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P672 = PInt<U672>; pub type N672 = NInt<U672>;
pub type U673 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P673 = PInt<U673>; pub type N673 = NInt<U673>;
pub type U674 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P674 = PInt<U674>; pub type N674 = NInt<U674>;
pub type U675 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P675 = PInt<U675>; pub type N675 = NInt<U675>;
pub type U676 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P676 = PInt<U676>; pub type N676 = NInt<U676>;
pub type U677 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P677 = PInt<U677>; pub type N677 = NInt<U677>;
pub type U678 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P678 = PInt<U678>; pub type N678 = NInt<U678>;
pub type U679 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P679 = PInt<U679>; pub type N679 = NInt<U679>;
pub type U680 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P680 = PInt<U680>; pub type N680 = NInt<U680>;
pub type U681 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P681 = PInt<U681>; pub type N681 = NInt<U681>;
pub type U682 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P682 = PInt<U682>; pub type N682 = NInt<U682>;
pub type U683 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P683 = PInt<U683>; pub type N683 = NInt<U683>;
pub type U684 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P684 = PInt<U684>; pub type N684 = NInt<U684>;
pub type U685 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P685 = PInt<U685>; pub type N685 = NInt<U685>;
pub type U686 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P686 = PInt<U686>; pub type N686 = NInt<U686>;
pub type U687 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P687 = PInt<U687>; pub type N687 = NInt<U687>;
pub type U688 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P688 = PInt<U688>; pub type N688 = NInt<U688>;
pub type U689 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P689 = PInt<U689>; pub type N689 = NInt<U689>;
pub type U690 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P690 = PInt<U690>; pub type N690 = NInt<U690>;
pub type U691 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P691 = PInt<U691>; pub type N691 = NInt<U691>;
pub type U692 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P692 = PInt<U692>; pub type N692 = NInt<U692>;
pub type U693 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P693 = PInt<U693>; pub type N693 = NInt<U693>;
pub type U694 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P694 = PInt<U694>; pub type N694 = NInt<U694>;
pub type U695 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P695 = PInt<U695>; pub type N695 = NInt<U695>;
pub type U696 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P696 = PInt<U696>; pub type N696 = NInt<U696>;
pub type U697 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P697 = PInt<U697>; pub type N697 = NInt<U697>;
pub type U698 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P698 = PInt<U698>; pub type N698 = NInt<U698>;
pub type U699 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P699 = PInt<U699>; pub type N699 = NInt<U699>;
pub type U700 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P700 = PInt<U700>; pub type N700 = NInt<U700>;
pub type U701 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P701 = PInt<U701>; pub type N701 = NInt<U701>;
pub type U702 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P702 = PInt<U702>; pub type N702 = NInt<U702>;
pub type U703 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P703 = PInt<U703>; pub type N703 = NInt<U703>;
pub type U704 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P704 = PInt<U704>; pub type N704 = NInt<U704>;
pub type U705 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P705 = PInt<U705>; pub type N705 = NInt<U705>;
pub type U706 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P706 = PInt<U706>; pub type N706 = NInt<U706>;
pub type U707 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P707 = PInt<U707>; pub type N707 = NInt<U707>;
pub type U708 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P708 = PInt<U708>; pub type N708 = NInt<U708>;
pub type U709 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P709 = PInt<U709>; pub type N709 = NInt<U709>;
pub type U710 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P710 = PInt<U710>; pub type N710 = NInt<U710>;
pub type U711 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P711 = PInt<U711>; pub type N711 = NInt<U711>;
pub type U712 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P712 = PInt<U712>; pub type N712 = NInt<U712>;
pub type U713 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P713 = PInt<U713>; pub type N713 = NInt<U713>;
pub type U714 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P714 = PInt<U714>; pub type N714 = NInt<U714>;
pub type U715 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P715 = PInt<U715>; pub type N715 = NInt<U715>;
pub type U716 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P716 = PInt<U716>; pub type N716 = NInt<U716>;
pub type U717 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P717 = PInt<U717>; pub type N717 = NInt<U717>;
pub type U718 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P718 = PInt<U718>; pub type N718 = NInt<U718>;
pub type U719 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P719 = PInt<U719>; pub type N719 = NInt<U719>;
pub type U720 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P720 = PInt<U720>; pub type N720 = NInt<U720>;
pub type U721 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P721 = PInt<U721>; pub type N721 = NInt<U721>;
pub type U722 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P722 = PInt<U722>; pub type N722 = NInt<U722>;
pub type U723 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P723 = PInt<U723>; pub type N723 = NInt<U723>;
pub type U724 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P724 = PInt<U724>; pub type N724 = NInt<U724>;
pub type U725 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P725 = PInt<U725>; pub type N725 = NInt<U725>;
pub type U726 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P726 = PInt<U726>; pub type N726 = NInt<U726>;
pub type U727 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P727 = PInt<U727>; pub type N727 = NInt<U727>;
pub type U728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P728 = PInt<U728>; pub type N728 = NInt<U728>;
pub type U729 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P729 = PInt<U729>; pub type N729 = NInt<U729>;
pub type U730 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P730 = PInt<U730>; pub type N730 = NInt<U730>;
pub type U731 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P731 = PInt<U731>; pub type N731 = NInt<U731>;
pub type U732 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P732 = PInt<U732>; pub type N732 = NInt<U732>;
pub type U733 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P733 = PInt<U733>; pub type N733 = NInt<U733>;
pub type U734 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P734 = PInt<U734>; pub type N734 = NInt<U734>;
pub type U735 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P735 = PInt<U735>; pub type N735 = NInt<U735>;
pub type U736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P736 = PInt<U736>; pub type N736 = NInt<U736>;
pub type U737 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P737 = PInt<U737>; pub type N737 = NInt<U737>;
pub type U738 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P738 = PInt<U738>; pub type N738 = NInt<U738>;
pub type U739 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P739 = PInt<U739>; pub type N739 = NInt<U739>;
pub type U740 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P740 = PInt<U740>; pub type N740 = NInt<U740>;
pub type U741 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P741 = PInt<U741>; pub type N741 = NInt<U741>;
pub type U742 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P742 = PInt<U742>; pub type N742 = NInt<U742>;
pub type U743 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P743 = PInt<U743>; pub type N743 = NInt<U743>;
pub type U744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P744 = PInt<U744>; pub type N744 = NInt<U744>;
pub type U745 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P745 = PInt<U745>; pub type N745 = NInt<U745>;
pub type U746 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P746 = PInt<U746>; pub type N746 = NInt<U746>;
pub type U747 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P747 = PInt<U747>; pub type N747 = NInt<U747>;
pub type U748 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P748 = PInt<U748>; pub type N748 = NInt<U748>;
pub type U749 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P749 = PInt<U749>; pub type N749 = NInt<U749>;
pub type U750 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P750 = PInt<U750>; pub type N750 = NInt<U750>;
pub type U751 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P751 = PInt<U751>; pub type N751 = NInt<U751>;
pub type U752 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P752 = PInt<U752>; pub type N752 = NInt<U752>;
pub type U753 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P753 = PInt<U753>; pub type N753 = NInt<U753>;
pub type U754 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P754 = PInt<U754>; pub type N754 = NInt<U754>;
pub type U755 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P755 = PInt<U755>; pub type N755 = NInt<U755>;
pub type U756 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P756 = PInt<U756>; pub type N756 = NInt<U756>;
pub type U757 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P757 = PInt<U757>; pub type N757 = NInt<U757>;
pub type U758 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P758 = PInt<U758>; pub type N758 = NInt<U758>;
pub type U759 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P759 = PInt<U759>; pub type N759 = NInt<U759>;
pub type U760 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P760 = PInt<U760>; pub type N760 = NInt<U760>;
pub type U761 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P761 = PInt<U761>; pub type N761 = NInt<U761>;
pub type U762 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P762 = PInt<U762>; pub type N762 = NInt<U762>;
pub type U763 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P763 = PInt<U763>; pub type N763 = NInt<U763>;
pub type U764 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P764 = PInt<U764>; pub type N764 = NInt<U764>;
pub type U765 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P765 = PInt<U765>; pub type N765 = NInt<U765>;
pub type U766 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P766 = PInt<U766>; pub type N766 = NInt<U766>;
pub type U767 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P767 = PInt<U767>; pub type N767 = NInt<U767>;
pub type U768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P768 = PInt<U768>; pub type N768 = NInt<U768>;
pub type U769 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P769 = PInt<U769>; pub type N769 = NInt<U769>;
pub type U770 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P770 = PInt<U770>; pub type N770 = NInt<U770>;
pub type U771 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P771 = PInt<U771>; pub type N771 = NInt<U771>;
pub type U772 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P772 = PInt<U772>; pub type N772 = NInt<U772>;
pub type U773 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P773 = PInt<U773>; pub type N773 = NInt<U773>;
pub type U774 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P774 = PInt<U774>; pub type N774 = NInt<U774>;
pub type U775 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P775 = PInt<U775>; pub type N775 = NInt<U775>;
pub type U776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P776 = PInt<U776>; pub type N776 = NInt<U776>;
pub type U777 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P777 = PInt<U777>; pub type N777 = NInt<U777>;
pub type U778 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P778 = PInt<U778>; pub type N778 = NInt<U778>;
pub type U779 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P779 = PInt<U779>; pub type N779 = NInt<U779>;
pub type U780 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P780 = PInt<U780>; pub type N780 = NInt<U780>;
pub type U781 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P781 = PInt<U781>; pub type N781 = NInt<U781>;
pub type U782 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P782 = PInt<U782>; pub type N782 = NInt<U782>;
pub type U783 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P783 = PInt<U783>; pub type N783 = NInt<U783>;
pub type U784 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P784 = PInt<U784>; pub type N784 = NInt<U784>;
pub type U785 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P785 = PInt<U785>; pub type N785 = NInt<U785>;
pub type U786 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P786 = PInt<U786>; pub type N786 = NInt<U786>;
pub type U787 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P787 = PInt<U787>; pub type N787 = NInt<U787>;
pub type U788 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P788 = PInt<U788>; pub type N788 = NInt<U788>;
pub type U789 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P789 = PInt<U789>; pub type N789 = NInt<U789>;
pub type U790 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P790 = PInt<U790>; pub type N790 = NInt<U790>;
pub type U791 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P791 = PInt<U791>; pub type N791 = NInt<U791>;
pub type U792 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P792 = PInt<U792>; pub type N792 = NInt<U792>;
pub type U793 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P793 = PInt<U793>; pub type N793 = NInt<U793>;
pub type U794 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P794 = PInt<U794>; pub type N794 = NInt<U794>;
pub type U795 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P795 = PInt<U795>; pub type N795 = NInt<U795>;
pub type U796 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P796 = PInt<U796>; pub type N796 = NInt<U796>;
pub type U797 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P797 = PInt<U797>; pub type N797 = NInt<U797>;
pub type U798 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P798 = PInt<U798>; pub type N798 = NInt<U798>;
pub type U799 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P799 = PInt<U799>; pub type N799 = NInt<U799>;
pub type U800 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P800 = PInt<U800>; pub type N800 = NInt<U800>;
pub type U801 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P801 = PInt<U801>; pub type N801 = NInt<U801>;
pub type U802 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P802 = PInt<U802>; pub type N802 = NInt<U802>;
pub type U803 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P803 = PInt<U803>; pub type N803 = NInt<U803>;
pub type U804 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P804 = PInt<U804>; pub type N804 = NInt<U804>;
pub type U805 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P805 = PInt<U805>; pub type N805 = NInt<U805>;
pub type U806 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P806 = PInt<U806>; pub type N806 = NInt<U806>;
pub type U807 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P807 = PInt<U807>; pub type N807 = NInt<U807>;
pub type U808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P808 = PInt<U808>; pub type N808 = NInt<U808>;
pub type U809 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P809 = PInt<U809>; pub type N809 = NInt<U809>;
pub type U810 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P810 = PInt<U810>; pub type N810 = NInt<U810>;
pub type U811 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P811 = PInt<U811>; pub type N811 = NInt<U811>;
pub type U812 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P812 = PInt<U812>; pub type N812 = NInt<U812>;
pub type U813 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P813 = PInt<U813>; pub type N813 = NInt<U813>;
pub type U814 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P814 = PInt<U814>; pub type N814 = NInt<U814>;
pub type U815 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P815 = PInt<U815>; pub type N815 = NInt<U815>;
pub type U816 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P816 = PInt<U816>; pub type N816 = NInt<U816>;
pub type U817 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P817 = PInt<U817>; pub type N817 = NInt<U817>;
pub type U818 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P818 = PInt<U818>; pub type N818 = NInt<U818>;
pub type U819 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P819 = PInt<U819>; pub type N819 = NInt<U819>;
pub type U820 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P820 = PInt<U820>; pub type N820 = NInt<U820>;
pub type U821 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P821 = PInt<U821>; pub type N821 = NInt<U821>;
pub type U822 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P822 = PInt<U822>; pub type N822 = NInt<U822>;
pub type U823 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P823 = PInt<U823>; pub type N823 = NInt<U823>;
pub type U824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P824 = PInt<U824>; pub type N824 = NInt<U824>;
pub type U825 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P825 = PInt<U825>; pub type N825 = NInt<U825>;
pub type U826 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P826 = PInt<U826>; pub type N826 = NInt<U826>;
pub type U827 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P827 = PInt<U827>; pub type N827 = NInt<U827>;
pub type U828 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P828 = PInt<U828>; pub type N828 = NInt<U828>;
pub type U829 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P829 = PInt<U829>; pub type N829 = NInt<U829>;
pub type U830 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P830 = PInt<U830>; pub type N830 = NInt<U830>;
pub type U831 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P831 = PInt<U831>; pub type N831 = NInt<U831>;
pub type U832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P832 = PInt<U832>; pub type N832 = NInt<U832>;
pub type U833 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P833 = PInt<U833>; pub type N833 = NInt<U833>;
pub type U834 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P834 = PInt<U834>; pub type N834 = NInt<U834>;
pub type U835 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P835 = PInt<U835>; pub type N835 = NInt<U835>;
pub type U836 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P836 = PInt<U836>; pub type N836 = NInt<U836>;
pub type U837 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P837 = PInt<U837>; pub type N837 = NInt<U837>;
pub type U838 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P838 = PInt<U838>; pub type N838 = NInt<U838>;
pub type U839 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P839 = PInt<U839>; pub type N839 = NInt<U839>;
pub type U840 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P840 = PInt<U840>; pub type N840 = NInt<U840>;
pub type U841 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P841 = PInt<U841>; pub type N841 = NInt<U841>;
pub type U842 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P842 = PInt<U842>; pub type N842 = NInt<U842>;
pub type U843 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P843 = PInt<U843>; pub type N843 = NInt<U843>;
pub type U844 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P844 = PInt<U844>; pub type N844 = NInt<U844>;
pub type U845 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P845 = PInt<U845>; pub type N845 = NInt<U845>;
pub type U846 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P846 = PInt<U846>; pub type N846 = NInt<U846>;
pub type U847 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P847 = PInt<U847>; pub type N847 = NInt<U847>;
pub type U848 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P848 = PInt<U848>; pub type N848 = NInt<U848>;
pub type U849 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P849 = PInt<U849>; pub type N849 = NInt<U849>;
pub type U850 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P850 = PInt<U850>; pub type N850 = NInt<U850>;
pub type U851 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P851 = PInt<U851>; pub type N851 = NInt<U851>;
pub type U852 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P852 = PInt<U852>; pub type N852 = NInt<U852>;
pub type U853 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P853 = PInt<U853>; pub type N853 = NInt<U853>;
pub type U854 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P854 = PInt<U854>; pub type N854 = NInt<U854>;
pub type U855 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P855 = PInt<U855>; pub type N855 = NInt<U855>;
pub type U856 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P856 = PInt<U856>; pub type N856 = NInt<U856>;
pub type U857 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P857 = PInt<U857>; pub type N857 = NInt<U857>;
pub type U858 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P858 = PInt<U858>; pub type N858 = NInt<U858>;
pub type U859 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P859 = PInt<U859>; pub type N859 = NInt<U859>;
pub type U860 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P860 = PInt<U860>; pub type N860 = NInt<U860>;
pub type U861 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P861 = PInt<U861>; pub type N861 = NInt<U861>;
pub type U862 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P862 = PInt<U862>; pub type N862 = NInt<U862>;
pub type U863 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P863 = PInt<U863>; pub type N863 = NInt<U863>;
pub type U864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P864 = PInt<U864>; pub type N864 = NInt<U864>;
pub type U865 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P865 = PInt<U865>; pub type N865 = NInt<U865>;
pub type U866 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P866 = PInt<U866>; pub type N866 = NInt<U866>;
pub type U867 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P867 = PInt<U867>; pub type N867 = NInt<U867>;
pub type U868 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P868 = PInt<U868>; pub type N868 = NInt<U868>;
pub type U869 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P869 = PInt<U869>; pub type N869 = NInt<U869>;
pub type U870 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P870 = PInt<U870>; pub type N870 = NInt<U870>;
pub type U871 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P871 = PInt<U871>; pub type N871 = NInt<U871>;
pub type U872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P872 = PInt<U872>; pub type N872 = NInt<U872>;
pub type U873 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P873 = PInt<U873>; pub type N873 = NInt<U873>;
pub type U874 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P874 = PInt<U874>; pub type N874 = NInt<U874>;
pub type U875 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P875 = PInt<U875>; pub type N875 = NInt<U875>;
pub type U876 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P876 = PInt<U876>; pub type N876 = NInt<U876>;
pub type U877 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P877 = PInt<U877>; pub type N877 = NInt<U877>;
pub type U878 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P878 = PInt<U878>; pub type N878 = NInt<U878>;
pub type U879 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P879 = PInt<U879>; pub type N879 = NInt<U879>;
pub type U880 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P880 = PInt<U880>; pub type N880 = NInt<U880>;
pub type U881 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P881 = PInt<U881>; pub type N881 = NInt<U881>;
pub type U882 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P882 = PInt<U882>; pub type N882 = NInt<U882>;
pub type U883 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P883 = PInt<U883>; pub type N883 = NInt<U883>;
pub type U884 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P884 = PInt<U884>; pub type N884 = NInt<U884>;
pub type U885 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P885 = PInt<U885>; pub type N885 = NInt<U885>;
pub type U886 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P886 = PInt<U886>; pub type N886 = NInt<U886>;
pub type U887 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P887 = PInt<U887>; pub type N887 = NInt<U887>;
pub type U888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P888 = PInt<U888>; pub type N888 = NInt<U888>;
pub type U889 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P889 = PInt<U889>; pub type N889 = NInt<U889>;
pub type U890 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P890 = PInt<U890>; pub type N890 = NInt<U890>;
pub type U891 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P891 = PInt<U891>; pub type N891 = NInt<U891>;
pub type U892 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P892 = PInt<U892>; pub type N892 = NInt<U892>;
pub type U893 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P893 = PInt<U893>; pub type N893 = NInt<U893>;
pub type U894 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P894 = PInt<U894>; pub type N894 = NInt<U894>;
pub type U895 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P895 = PInt<U895>; pub type N895 = NInt<U895>;
pub type U896 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P896 = PInt<U896>; pub type N896 = NInt<U896>;
pub type U897 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P897 = PInt<U897>; pub type N897 = NInt<U897>;
pub type U898 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P898 = PInt<U898>; pub type N898 = NInt<U898>;
pub type U899 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P899 = PInt<U899>; pub type N899 = NInt<U899>;
pub type U900 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P900 = PInt<U900>; pub type N900 = NInt<U900>;
pub type U901 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P901 = PInt<U901>; pub type N901 = NInt<U901>;
pub type U902 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P902 = PInt<U902>; pub type N902 = NInt<U902>;
pub type U903 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P903 = PInt<U903>; pub type N903 = NInt<U903>;
pub type U904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P904 = PInt<U904>; pub type N904 = NInt<U904>;
pub type U905 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P905 = PInt<U905>; pub type N905 = NInt<U905>;
pub type U906 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P906 = PInt<U906>; pub type N906 = NInt<U906>;
pub type U907 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P907 = PInt<U907>; pub type N907 = NInt<U907>;
pub type U908 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P908 = PInt<U908>; pub type N908 = NInt<U908>;
pub type U909 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P909 = PInt<U909>; pub type N909 = NInt<U909>;
pub type U910 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P910 = PInt<U910>; pub type N910 = NInt<U910>;
pub type U911 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P911 = PInt<U911>; pub type N911 = NInt<U911>;
pub type U912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P912 = PInt<U912>; pub type N912 = NInt<U912>;
pub type U913 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P913 = PInt<U913>; pub type N913 = NInt<U913>;
pub type U914 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P914 = PInt<U914>; pub type N914 = NInt<U914>;
pub type U915 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P915 = PInt<U915>; pub type N915 = NInt<U915>;
pub type U916 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P916 = PInt<U916>; pub type N916 = NInt<U916>;
pub type U917 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P917 = PInt<U917>; pub type N917 = NInt<U917>;
pub type U918 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P918 = PInt<U918>; pub type N918 = NInt<U918>;
pub type U919 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P919 = PInt<U919>; pub type N919 = NInt<U919>;
pub type U920 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P920 = PInt<U920>; pub type N920 = NInt<U920>;
pub type U921 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P921 = PInt<U921>; pub type N921 = NInt<U921>;
pub type U922 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P922 = PInt<U922>; pub type N922 = NInt<U922>;
pub type U923 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P923 = PInt<U923>; pub type N923 = NInt<U923>;
pub type U924 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P924 = PInt<U924>; pub type N924 = NInt<U924>;
pub type U925 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P925 = PInt<U925>; pub type N925 = NInt<U925>;
pub type U926 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P926 = PInt<U926>; pub type N926 = NInt<U926>;
pub type U927 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P927 = PInt<U927>; pub type N927 = NInt<U927>;
pub type U928 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P928 = PInt<U928>; pub type N928 = NInt<U928>;
pub type U929 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P929 = PInt<U929>; pub type N929 = NInt<U929>;
pub type U930 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P930 = PInt<U930>; pub type N930 = NInt<U930>;
pub type U931 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P931 = PInt<U931>; pub type N931 = NInt<U931>;
pub type U932 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P932 = PInt<U932>; pub type N932 = NInt<U932>;
pub type U933 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P933 = PInt<U933>; pub type N933 = NInt<U933>;
pub type U934 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P934 = PInt<U934>; pub type N934 = NInt<U934>;
pub type U935 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P935 = PInt<U935>; pub type N935 = NInt<U935>;
pub type U936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P936 = PInt<U936>; pub type N936 = NInt<U936>;
pub type U937 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P937 = PInt<U937>; pub type N937 = NInt<U937>;
pub type U938 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P938 = PInt<U938>; pub type N938 = NInt<U938>;
pub type U939 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P939 = PInt<U939>; pub type N939 = NInt<U939>;
pub type U940 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P940 = PInt<U940>; pub type N940 = NInt<U940>;
pub type U941 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P941 = PInt<U941>; pub type N941 = NInt<U941>;
pub type U942 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P942 = PInt<U942>; pub type N942 = NInt<U942>;
pub type U943 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P943 = PInt<U943>; pub type N943 = NInt<U943>;
pub type U944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P944 = PInt<U944>; pub type N944 = NInt<U944>;
pub type U945 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P945 = PInt<U945>; pub type N945 = NInt<U945>;
pub type U946 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P946 = PInt<U946>; pub type N946 = NInt<U946>;
pub type U947 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P947 = PInt<U947>; pub type N947 = NInt<U947>;
pub type U948 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P948 = PInt<U948>; pub type N948 = NInt<U948>;
pub type U949 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P949 = PInt<U949>; pub type N949 = NInt<U949>;
pub type U950 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P950 = PInt<U950>; pub type N950 = NInt<U950>;
pub type U951 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P951 = PInt<U951>; pub type N951 = NInt<U951>;
pub type U952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P952 = PInt<U952>; pub type N952 = NInt<U952>;
pub type U953 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P953 = PInt<U953>; pub type N953 = NInt<U953>;
pub type U954 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P954 = PInt<U954>; pub type N954 = NInt<U954>;
pub type U955 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P955 = PInt<U955>; pub type N955 = NInt<U955>;
pub type U956 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P956 = PInt<U956>; pub type N956 = NInt<U956>;
pub type U957 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P957 = PInt<U957>; pub type N957 = NInt<U957>;
pub type U958 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P958 = PInt<U958>; pub type N958 = NInt<U958>;
pub type U959 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P959 = PInt<U959>; pub type N959 = NInt<U959>;
pub type U960 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P960 = PInt<U960>; pub type N960 = NInt<U960>;
pub type U961 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P961 = PInt<U961>; pub type N961 = NInt<U961>;
pub type U962 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P962 = PInt<U962>; pub type N962 = NInt<U962>;
pub type U963 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P963 = PInt<U963>; pub type N963 = NInt<U963>;
pub type U964 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P964 = PInt<U964>; pub type N964 = NInt<U964>;
pub type U965 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P965 = PInt<U965>; pub type N965 = NInt<U965>;
pub type U966 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P966 = PInt<U966>; pub type N966 = NInt<U966>;
pub type U967 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P967 = PInt<U967>; pub type N967 = NInt<U967>;
pub type U968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P968 = PInt<U968>; pub type N968 = NInt<U968>;
pub type U969 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P969 = PInt<U969>; pub type N969 = NInt<U969>;
pub type U970 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P970 = PInt<U970>; pub type N970 = NInt<U970>;
pub type U971 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P971 = PInt<U971>; pub type N971 = NInt<U971>;
pub type U972 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P972 = PInt<U972>; pub type N972 = NInt<U972>;
pub type U973 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P973 = PInt<U973>; pub type N973 = NInt<U973>;
pub type U974 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P974 = PInt<U974>; pub type N974 = NInt<U974>;
pub type U975 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P975 = PInt<U975>; pub type N975 = NInt<U975>;
pub type U976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P976 = PInt<U976>; pub type N976 = NInt<U976>;
pub type U977 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P977 = PInt<U977>; pub type N977 = NInt<U977>;
pub type U978 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P978 = PInt<U978>; pub type N978 = NInt<U978>;
pub type U979 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P979 = PInt<U979>; pub type N979 = NInt<U979>;
pub type U980 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P980 = PInt<U980>; pub type N980 = NInt<U980>;
pub type U981 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P981 = PInt<U981>; pub type N981 = NInt<U981>;
pub type U982 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P982 = PInt<U982>; pub type N982 = NInt<U982>;
pub type U983 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P983 = PInt<U983>; pub type N983 = NInt<U983>;
pub type U984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P984 = PInt<U984>; pub type N984 = NInt<U984>;
pub type U985 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P985 = PInt<U985>; pub type N985 = NInt<U985>;
pub type U986 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P986 = PInt<U986>; pub type N986 = NInt<U986>;
pub type U987 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P987 = PInt<U987>; pub type N987 = NInt<U987>;
pub type U988 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P988 = PInt<U988>; pub type N988 = NInt<U988>;
pub type U989 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P989 = PInt<U989>; pub type N989 = NInt<U989>;
pub type U990 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P990 = PInt<U990>; pub type N990 = NInt<U990>;
pub type U991 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P991 = PInt<U991>; pub type N991 = NInt<U991>;
pub type U992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P992 = PInt<U992>; pub type N992 = NInt<U992>;
pub type U993 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P993 = PInt<U993>; pub type N993 = NInt<U993>;
pub type U994 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P994 = PInt<U994>; pub type N994 = NInt<U994>;
pub type U995 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P995 = PInt<U995>; pub type N995 = NInt<U995>;
pub type U996 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P996 = PInt<U996>; pub type N996 = NInt<U996>;
pub type U997 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P997 = PInt<U997>; pub type N997 = NInt<U997>;
pub type U998 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P998 = PInt<U998>; pub type N998 = NInt<U998>;
pub type U999 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P999 = PInt<U999>; pub type N999 = NInt<U999>;
pub type U1000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1000 = PInt<U1000>; pub type N1000 = NInt<U1000>;
pub type U1001 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1001 = PInt<U1001>; pub type N1001 = NInt<U1001>;
pub type U1002 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1002 = PInt<U1002>; pub type N1002 = NInt<U1002>;
pub type U1003 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1003 = PInt<U1003>; pub type N1003 = NInt<U1003>;
pub type U1004 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1004 = PInt<U1004>; pub type N1004 = NInt<U1004>;
pub type U1005 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1005 = PInt<U1005>; pub type N1005 = NInt<U1005>;
pub type U1006 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1006 = PInt<U1006>; pub type N1006 = NInt<U1006>;
pub type U1007 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1007 = PInt<U1007>; pub type N1007 = NInt<U1007>;
pub type U1008 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
```

```
pub type P1008 = PInt<U1008>; pub type N1008 = NInt<U1008>;
pub type U1009 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1009 = PInt<U1009>; pub type N1009 = NInt<U1009>;
pub type U1010 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1010 = PInt<U1010>; pub type N1010 = NInt<U1010>;
pub type U1011 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1011 = PInt<U1011>; pub type N1011 = NInt<U1011>;
pub type U1012 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1012 = PInt<U1012>; pub type N1012 = NInt<U1012>;
pub type U1013 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1013 = PInt<U1013>; pub type N1013 = NInt<U1013>;
pub type U1014 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1014 = PInt<U1014>; pub type N1014 = NInt<U1014>;
pub type U1015 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1015 = PInt<U1015>; pub type N1015 = NInt<U1015>;
pub type U1016 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1016 = PInt<U1016>; pub type N1016 = NInt<U1016>;
pub type U1017 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1017 = PInt<U1017>; pub type N1017 = NInt<U1017>;
pub type U1018 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1018 = PInt<U1018>; pub type N1018 = NInt<U1018>;
pub type U1019 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1019 = PInt<U1019>; pub type N1019 = NInt<U1019>;
pub type U1020 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1020 = PInt<U1020>; pub type N1020 = NInt<U1020>;
pub type U1021 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1021 = PInt<U1021>; pub type N1021 = NInt<U1021>;
pub type U1022 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1022 = PInt<U1022>; pub type N1022 = NInt<U1022>;
pub type U1023 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1023 = PInt<U1023>; pub type N1023 = NInt<U1023>;
pub type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1024 = PInt<U1024>; pub type N1024 = NInt<U1024>;
pub type U2048 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2048 = PInt<U2048>; pub type N2048 = NInt<U2048>;
pub type U4096 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4096 = PInt<U4096>; pub type N4096 = NInt<U4096>;
pub type U8192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P8192 = PInt<U8192>; pub type N8192 = NInt<U8192>;
pub type U16384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P16384 = PInt<U16384>; pub type N16384 = NInt<U16384>;
pub type U32768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P32768 = PInt<U32768>; pub type N32768 = NInt<U32768>;
pub type U65536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P65536 = PInt<U65536>; pub type N65536 = NInt<U65536>;
pub type U131072 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P131072 = PInt<U131072>; pub type N131072 = NInt<U131072>;
pub type U262144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P262144 = PInt<U262144>; pub type N262144 = NInt<U262144>;
pub type U524288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P524288 = PInt<U524288>; pub type N524288 = NInt<U524288>;
pub type U1048576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
```

```
pub type P1048576 = PInt<U1048576>; pub type N1048576 = NInt<U1048576>;
pub type U2097152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P2097152 = PInt<U2097152>; pub type N2097152 = NInt<U2097152>;
pub type U4194304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P4194304 = PInt<U4194304>; pub type N4194304 = NInt<U4194304>;
pub type U8388608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P8388608 = PInt<U8388608>; pub type N8388608 = NInt<U8388608>;
pub type U16777216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P16777216 = PInt<U16777216>; pub type N16777216 = NInt<U167772
pub type U33554432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P33554432 = PInt<U33554432>; pub type N33554432 = NInt<U335544
pub type U67108864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P67108864 = PInt<U67108864>; pub type N67108864 = NInt<U671088
pub type U134217728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P134217728 = PInt<U134217728>; pub type N134217728 = NInt<U134
pub type U268435456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P268435456 = PInt<U268435456>; pub type N268435456 = NInt<U268
pub type U536870912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P536870912 = PInt<U536870912>; pub type N536870912 = NInt<U536
pub type U1073741824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1073741824 = PInt<U1073741824>; pub type N1073741824 = NInt<U
pub type U2147483648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P2147483648 = PInt<U2147483648>; pub type N2147483648 = NInt<U
pub type U4294967296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P4294967296 = PInt<U4294967296>; pub type N4294967296 = NInt<U
pub type U8589934592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P8589934592 = PInt<U8589934592>; pub type N8589934592 = NInt<U
pub type U17179869184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P17179869184 = PInt<U17179869184>; pub type N17179869184 = NIn
pub type U34359738368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P34359738368 = PInt<U34359738368>; pub type N34359738368 = NIn
pub type U68719476736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P68719476736 = PInt<U68719476736>; pub type N68719476736 = NIn
pub type U137438953472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P137438953472 = PInt<U137438953472>; pub type N137438953472 =
pub type U274877906944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P274877906944 = PInt<U274877906944>; pub type N274877906944 =
pub type U549755813888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P549755813888 = PInt<U549755813888>; pub type N549755813888 =
pub type U1099511627776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1099511627776 = PInt<U1099511627776>; pub type N1099511627776
pub type U2199023255552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P2199023255552 = PInt<U2199023255552>; pub type N2199023255552
pub type U4398046511104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P4398046511104 = PInt<U4398046511104>; pub type N4398046511104
pub type U8796093022208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P8796093022208 = PInt<U8796093022208>; pub type N8796093022208
pub type U17592186044416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P17592186044416 = PInt<U17592186044416>; pub type N17592186044
pub type U35184372088832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P35184372088832 = PInt<U35184372088832>; pub type N35184372088
pub type U70368744177664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
```

```
pub type P70368744177664 = PInt<U70368744177664>; pub type N7036874417
pub type U140737488355328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P140737488355328 = PInt<U140737488355328>; pub type N140737488
pub type U281474976710656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P281474976710656 = PInt<U281474976710656>; pub type N281474976
pub type U562949953421312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P562949953421312 = PInt<U562949953421312>; pub type N562949953
pub type U1125899906842624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P1125899906842624 = PInt<U1125899906842624>; pub type N1125899
pub type U2251799813685248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P2251799813685248 = PInt<U2251799813685248>; pub type N2251799
pub type U4503599627370496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P4503599627370496 = PInt<U4503599627370496>; pub type N4503599
pub type U9007199254740992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P9007199254740992 = PInt<U9007199254740992>; pub type N9007199
pub type U18014398509481984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P18014398509481984 = PInt<U18014398509481984>; pub type N18014
pub type U36028797018963968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P36028797018963968 = PInt<U36028797018963968>; pub type N36028
pub type U72057594037927936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P72057594037927936 = PInt<U72057594037927936>; pub type N72057
pub type U144115188075855872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P144115188075855872 = PInt<U144115188075855872>; pub type N144
pub type U288230376151711744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P288230376151711744 = PInt<U288230376151711744>; pub type N288
pub type U576460752303423488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P576460752303423488 = PInt<U576460752303423488>; pub type N576
pub type U1152921504606846976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1152921504606846976 = PInt<U1152921504606846976>; pub type N1
pub type U2305843009213693952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2305843009213693952 = PInt<U2305843009213693952>; pub type N2
pub type U4611686018427387904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4611686018427387904 = PInt<U4611686018427387904>; pub type N4
pub type U9223372036854775808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type U10000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P10000 = PInt<U10000>; pub type N10000 = NInt<U10000>;
pub type U100000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P100000 = PInt<U100000>; pub type N100000 = NInt<U100000>;
pub type U1000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1000000 = PInt<U1000000>; pub type N1000000 = NInt<U1000000>;
pub type U10000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P10000000 = PInt<U10000000>; pub type N10000000 = NInt<U100000
pub type U100000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P100000000 = PInt<U100000000>; pub type N100000000 = NInt<U100
pub type U1000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1000000000 = PInt<U1000000000>; pub type N1000000000 = NInt<U
pub type U10000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P10000000000 = PInt<U10000000000>; pub type N10000000000 = NIn
pub type U100000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P100000000000 = PInt<U100000000000>; pub type N100000000000 =
pub type U1000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1000000000000 = PInt<U1000000000000>; pub type N1000000000000
```

```
    pub type U10000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P10000000000000 = PInt<U10000000000000>; pub type N10000000000
    pub type U100000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P100000000000000 = PInt<U100000000000000>; pub type N100000000
    pub type U1000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P1000000000000000 = PInt<U1000000000000000>; pub type N1000000
    pub type U10000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P10000000000000000 = PInt<U10000000000000000>; pub type N10000
    pub type U100000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P100000000000000000 = PInt<U100000000000000000>; pub type N100
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P1000000000000000000 = PInt<U1000000000000000000>; pub type N1
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
}
```

File: ./target/release/build/typenum-346c6668244bcd95/out/tests.rs

```
extern crate typenum;

use std::ops::*;
use std::cmp::Ordering;
use typenum::*;

#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitOrU0 = <<A as BitOr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitOrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_BitXor_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitXorU0 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitXorU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU0 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU0 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0AddU0 = <<A as Add<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0AddU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MaxU0 = <<A as Max<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MaxU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0GcdU0 = <<A as Gcd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0GcdU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Sub_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0SubU0 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0SubU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_0() {
    type A = UTerm;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_0() {
    type A = UTerm;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U0CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_1() {
    type A = UTerm;
```

```
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU1 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitXorU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU1 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU1 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0AddU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0MinU1 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU1 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU1 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU1 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU2 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitXorU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU2 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU2 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0AddU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU2 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_Mul_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU2 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0PowU2 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU3 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitXorU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU3 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU3 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0AddU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU3 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_3() {
    type A = UTerm;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU3 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU3 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_BitOr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU4 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitXorU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU4 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U0AddU4 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0AddU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU4 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU4 = <<A as Mul<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0MulU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU4 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_Cmp_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU5 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitXorU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0ShlU5 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU5 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0AddU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU5 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU5 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_5() {
    type A = UTerm;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU5 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU0 = <<A as BitOr<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U1BitOrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU0 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitXorU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShlU0 = <<A as Shl<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShlU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShrU0 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU0 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1AddU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Min_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU0 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU0 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1SubU0 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1SubU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U1CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitXorU1 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitXorU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU1 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1ShlU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU1 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1AddU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_1() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1SubU1 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1SubU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U1MulU1 = <<A as Mul<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MulU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1DivU1 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1DivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_PartialDiv_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U1PartialDivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU1 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU2 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitXorU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U1ShlU2 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1ShlU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU2 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1AddU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU2 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U1MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU2 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MulU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Pow_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU2 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U1BitXorU3 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1BitXorU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU3 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U1ShlU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU3 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1AddU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU3 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_Max_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU3 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MulU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_3() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU3 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U1BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU4 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitXorU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU4 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U1ShlU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU4 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1AddU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Min_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU4 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU4 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MulU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU4 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU5 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1BitXorU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U1ShlU5 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U1ShlU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_5() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU5 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U1AddU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU5 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U1MulU5 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MulU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU5 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU5 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU0 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU0 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitXorU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU0 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShlU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2ShrU0 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU0 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2AddU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU0 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU0 = <<A as Gcd<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2GcdU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2SubU0 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2SubU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U2CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU1 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitXorU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU1 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2ShlU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU1 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2AddU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU1 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2SubU1 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2SubU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU1 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MulU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2DivU1 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2DivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U2PartialDivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU1 = <<A as Pow<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2PowU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitXorU2 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitXorU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU2 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2ShlU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU2 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2AddU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2SubU2 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2SubU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2MulU2 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MulU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U2PartialDivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU2 = <<A as Pow<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U2PowU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU3 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU3 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2BitXorU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU3 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2ShlU3 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU3 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2AddU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU3 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU3 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2MulU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU3 = <<A as Pow<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2PowU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU4 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitOrU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2BitXorU4 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitXorU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU4 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2ShlU4 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU4 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2AddU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU4 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2MinU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU4 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU4 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2MulU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU4 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU4 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2PowU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU5 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitXorU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU5 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U2ShlU5 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU5 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2AddU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU5 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU5 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U2MulU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU5 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2PowU5 = <<A as Pow<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2PowU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU0 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU0 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitXorU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShlU0 = <<A as Shl<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShlU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU0 = <<A as Shr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU0 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3AddU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU0 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU0 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3SubU0 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3SubU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U3CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU1 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitXorU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU1 = <<A as Shl<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3ShlU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU1 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3AddU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU1 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3SubU1 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3SubU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU1 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MulU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU1 = <<A as Div<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3DivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U3>>::Outpu

    assert_eq!(<U3PartialDivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU1 = <<A as Pow<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3PowU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3BitXorU2 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitXorU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU2 = <<A as Shl<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3ShlU2 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU2 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3AddU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U3MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU2 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU2 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3SubU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU2 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3MulU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU2 = <<A as Pow<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3PowU2 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U3BitAndU3 = <<A as BitAnd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitAndU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitXorU3 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitXorU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU3 = <<A as Shl<B>>::Output as Same<U24>>::Output;

    assert_eq!(<U3ShlU3 as Unsigned>::to_u64(), <U24 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU3 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3AddU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3SubU3 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3SubU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU3 = <<A as Mul<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3MulU3 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U3PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U3PartialDivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU3 = <<A as Pow<B>>::Output as Same<U27>>::Output;

    assert_eq!(<U3PowU3 as Unsigned>::to_u64(), <U27 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU4 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU4 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitXorU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U48 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU4 = <<A as Shl<B>>::Output as Same<U48>>::Output;

    assert_eq!(<U3ShlU4 as Unsigned>::to_u64(), <U48 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU4 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3AddU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3MinU4 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU4 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3MulU4 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU4 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U3PowU4 = <<A as Pow<B>>::Output as Same<U81>>::Output;

    assert_eq!(<U3PowU4 as Unsigned>::to_u64(), <U81 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU5 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3BitXorU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU5 = <<A as Shl<B>>::Output as Same<U96>>::Output;

    assert_eq!(<U3ShlU5 as Unsigned>::to_u64(), <U96 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3AddU5 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U3AddU5 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU5 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU5 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U3MulU5 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU5 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U3PowU5 = <<A as Pow<B>>::Output as Same<U243>>::Output;

    assert_eq!(<U3PowU5 as Unsigned>::to_u64(), <U243 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU0 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU0 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitXorU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU0 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShlU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU0 = <<A as Shr<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4ShrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU0 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4AddU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU0 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU0 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU0 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4SubU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U4CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U4BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU1 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitXorU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU1 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4ShlU1 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU1 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4AddU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU1 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU1 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4SubU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU1 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MulU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU1 = <<A as Div<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4DivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U4PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U4>>::Outpu

    assert_eq!(<U4PartialDivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU1 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4PowU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU2 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitOrU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU2 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitXorU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU2 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4ShlU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU2 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4AddU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU2 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU2 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4SubU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU2 = <<A as Mul<B>>::Output as Same<U8>>::Output;
```

```
    assert_eq!(<U4MulU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U4PartialDivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU2 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4PowU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU3 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitXorU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
```

```
    type U4ShlU3 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U4ShlU3 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU3 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4AddU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU3 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU3 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4SubU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU3 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U4MulU3 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4PowU3 = <<A as Pow<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4PowU3 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitXorU4 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitXorU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU4 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4ShlU4 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU4 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4AddU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4SubU4 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4SubU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MulU4 = <<A as Mul<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4MulU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U4PartialDivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0

    #[allow(non_camel_case_types)]
    type U4PowU4 = <<A as Pow<B>>::Output as Same<U256>>::Output;
```

```
    assert_eq!(<U4PowU4 as Unsigned>::to_u64(), <U256 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU5 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU5 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4BitXorU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0

    #[allow(non_camel_case_types)]
    type U4ShlU5 = <<A as Shl<B>>::Output as Same<U128>>::Output;

    assert_eq!(<U4ShlU5 as Unsigned>::to_u64(), <U128 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU5 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U4AddU5 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU5 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U4MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU5 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U4MulU5 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4RemU5 = <<A as Rem<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4RemU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTe

    #[allow(non_camel_case_types)]
    type U4PowU5 = <<A as Pow<B>>::Output as Same<U1024>>::Output;

    assert_eq!(<U4PowU5 as Unsigned>::to_u64(), <U1024 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU0 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitXorU0 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitXorU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShlU0 = <<A as Shl<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShlU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU0 = <<A as Shr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU0 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5AddU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U5MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU0 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU0 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU0 = <<A as Sub<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5SubU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U5CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U5BitXorU1 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitXorU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU1 = <<A as Shl<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5ShlU1 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU1 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5AddU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU1 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU1 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5SubU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU1 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MulU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU1 = <<A as Div<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5DivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U5>>::Outpu

    assert_eq!(<U5PartialDivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU1 = <<A as Pow<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5PowU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU1 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<U5CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU2 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU2 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitXorU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU2 = <<A as Shl<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5ShlU2 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU2 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5AddU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU2 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU2 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5SubU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU2 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5MulU2 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U5RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU2 = <<A as Pow<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5PowU2 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU3 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5BitXorU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU3 = <<A as Shl<B>>::Output as Same<U40>>::Output;

    assert_eq!(<U5ShlU3 as Unsigned>::to_u64(), <U40 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU3 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U5AddU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U5MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU3 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU3 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5SubU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU3 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U5MulU3 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U5PowU3 = <<A as Pow<B>>::Output as Same<U125>>::Output;

    assert_eq!(<U5PowU3 as Unsigned>::to_u64(), <U125 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU4 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitXorU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU4 = <<A as Shl<B>>::Output as Same<U80>>::Output;

    assert_eq!(<U5ShlU4 as Unsigned>::to_u64(), <U80 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U5ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU4 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U5AddU4 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU4 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU4 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5SubU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU4 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5MulU4 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1
```

```
    #[allow(non_camel_case_types)]
    type U5PowU4 = <<A as Pow<B>>::Output as Same<U625>>::Output;

    assert_eq!(<U5PowU4 as Unsigned>::to_u64(), <U625 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU5 = <<A as BitAnd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitAndU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitXorU5 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitXorU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1

    #[allow(non_camel_case_types)]
    type U5ShlU5 = <<A as Shl<B>>::Output as Same<U160>>::Output;

    assert_eq!(<U5ShlU5 as Unsigned>::to_u64(), <U160 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU5 = <<A as Add<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5AddU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU5 = <<A as Min<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MinU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5SubU5 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5SubU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U25 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU5 = <<A as Mul<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5MulU5 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U5DivU5 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U5PartialDivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type U5PowU5 = <<A as Pow<B>>::Output as Same<U3125>>::Output;

    assert_eq!(<U5PowU5 as Unsigned>::to_u64(), <U3125 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN5 = <<A as Add<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5AddN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5SubN5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5SubN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulN5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5MulN5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MaxN5 = <<A as Max<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MaxN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu
```

```
    assert_eq!(<N5PartialDivN5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN4 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5AddN4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5SubN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulN4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N5MulN4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN3 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5AddN3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5SubN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N5MulN3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N5RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN2 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5AddN2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5SubN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulN2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N5MulN2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN1 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5AddN1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5SubN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN1 = <<A as Mul<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<N5MulN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5DivN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<N5PartialDivN1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Add_0 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Add_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5Sub_0 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Sub_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Min_0 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Min_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N5Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5AddP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP1 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5SubP1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MulP1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MulP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP1 = <<A as Div<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5DivP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<N5PartialDivP1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PowP1 = <<A as Pow<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5PowP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5AddP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP2 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5SubP2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulP2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5MulP2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N5MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5AddP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP3 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5SubP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N5MulP3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MinP3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N5MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N5RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type N5PowP3 = <<A as Pow<B>>::Output as Same<N125>>::Output;

    assert_eq!(<N5PowP3 as Integer>::to_i64(), <N125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5AddP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP4 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5SubP4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulP4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N5MulP4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N5MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type N5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<N5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5AddP5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5AddP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP5 = <<A as Sub<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5SubP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulP5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<N5MulP5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N5PartialDivP5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N5PowP5 = <<A as Pow<B>>::Output as Same<N3125>>::Output;
```

```
    assert_eq!(<N5PowP5 as Integer>::to_i64(), <N3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN5 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4AddN5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN5 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4SubN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N4MulN5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN5 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N4RemN5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN4 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4AddN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4SubN4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4SubN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4MulN4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N4_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N4PartialDivN4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN3 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4AddN3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4SubN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N4MulN3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN2 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4AddN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubN2 = <<A as Sub<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N4SubN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N4MulN2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N4_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N4PartialDivN2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4AddN1 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4AddN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4SubN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MulN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4DivN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<N4PartialDivN1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Add_0 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Add_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Sub_0 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Sub_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Min_0 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4Min_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N4Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4AddP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP1 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4SubP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MulP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4DivP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<N4PartialDivP1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PowP1 = <<A as Pow<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4PowP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddP2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4AddP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP2 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4SubP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MulP2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4MulP2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N4DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N4PartialDivP2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4AddP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP3 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4SubP3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N4MulP3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N4MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N64 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP3 = <<A as Pow<B>>::Output as Same<N64>>::Output;

    assert_eq!(<N4PowP3 as Integer>::to_i64(), <N64 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4AddP4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4AddP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP4 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4SubP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<N4MulP4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MinP4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N4PartialDivP4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type N4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<N4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP5 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4AddP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP5 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4SubP5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N4MulP5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP5 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemP5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1024 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N4PowP5 = <<A as Pow<B>>::Output as Same<N1024>>::Output;

    assert_eq!(<N4PowP5 as Integer>::to_i64(), <N1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN5 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3AddN5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN5 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3SubN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N3MulN5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxN5 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN4 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3AddN4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3SubN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N3MulN4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN4 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN4 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<N3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN3 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3AddN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3SubN3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3SubN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3MulN3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N3PartialDivN3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN2 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3AddN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3SubN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N3MulN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3AddN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3SubN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN1 = <<A as Mul<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<N3MulN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3DivN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<N3PartialDivN1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Add_0 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Add_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N3Sub_0 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Sub_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Min_0 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Min_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N3Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3AddP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3SubP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MulP1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MulP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP1 = <<A as Div<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3DivP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<N3PartialDivP1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP1 = <<A as Pow<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3PowP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3AddP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP2 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3SubP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3MulP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N3MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3AddP3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3AddP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP3 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3SubP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N3MulP3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinP3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N3PartialDivP3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N27 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP3 = <<A as Pow<B>>::Output as Same<N27>>::Output;

    assert_eq!(<N3PowP3 as Integer>::to_i64(), <N27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3AddP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP4 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3SubP4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N3MulP4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP4 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<N3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP5 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3AddP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP5 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3SubP5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N3MulP5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP5 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N243 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type N3PowP5 = <<A as Pow<B>>::Output as Same<N243>>::Output;
```

```
    assert_eq!(<N3PowP5 as Integer>::to_i64(), <N243 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN5 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2AddN5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN5 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2SubN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N2MulN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN5 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N2RemN5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN4 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2AddN4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubN4 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2SubN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N2MulN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN4 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2RemN4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN3 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2AddN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2SubN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N2MulN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN3 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN2 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2AddN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2SubN2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2SubN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN2 = <<A as Mul<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<N2MulN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N2PartialDivN2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2AddN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2SubN1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2SubN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MulN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivN1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2DivN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N2PartialDivN1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2Add_0 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Add_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Sub_0 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Sub_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Min_0 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Min_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N2Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2AddP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2SubP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MulP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivP1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2DivP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N2PartialDivP1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP1 = <<A as Pow<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2PowP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2AddP2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2AddP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP2 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2SubP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MulP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MinP2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N2PartialDivP2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2AddP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP3 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2SubP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2MulP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP3 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP3 = <<A as Pow<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2PowP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddP4 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2AddP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP4 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2SubP4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2MulP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP4 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;
```

```
    assert_eq!(<N2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP5 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2AddP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP5 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2SubP5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N2MulP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP5 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N2RemP5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N32 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP5 = <<A as Pow<B>>::Output as Same<N32>>::Output;

    assert_eq!(<N2PowP5 as Integer>::to_i64(), <N32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN5 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1AddN5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN5 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1SubN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Mul_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MulN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN5 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN4 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1AddN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN4 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1SubN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MulN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN4 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N4() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN4 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<N1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN3 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1AddN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1SubN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MulN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Max_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN3 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1PowN3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1AddN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1SubN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MulN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN2 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N2() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1AddN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1SubN1 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N1SubN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MulN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Div_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivN1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1DivN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N1PartialDivN1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Add_0 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Add_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Sub_0 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Sub_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Min_0 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Min_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Max__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N1Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1AddP1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1AddP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1SubP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MulP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivP1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1DivP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N1PartialDivP1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Pow_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1AddP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1SubP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N1MulP2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MulP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP2 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1AddP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N1SubP3 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1SubP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MulP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP3 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P4() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP4 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1AddP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP4 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1SubP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MulP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP4 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP5 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1AddP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP5 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1SubP5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MulP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1MinP5 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP5 = <<A as Rem<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N1RemP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN5 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0AddN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN5 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0SubN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N5() {
    type A = Z0;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN5 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type _0DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN4 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0AddN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Sub_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN4 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0SubN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN4 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type _0GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN3 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0AddN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN3 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0SubN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N3() {
    type A = Z0;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN3 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type _0PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0AddN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0SubN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Min_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN2 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0AddN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0SubN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN1 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N1() {
    type A = Z0;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Add_0 = <<A as Add<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0Add_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Sub_0 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Sub_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Gcd__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Gcd_0 = <<A as Gcd<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Gcd_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow__0() {
    type A = Z0;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp__0() {
    type A = Z0;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type _0Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<_0Cmp_0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0AddP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0SubP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP1 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP1 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P1() {
    type A = Z0;
```

```
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0AddP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0SubP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP2 = <<A as Min<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MinP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_PartialDiv_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP2 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP3 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0AddP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP3 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0SubP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP3 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP3 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P3() {
    type A = Z0;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP4 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0AddP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP4 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0SubP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP4 = <<A as Min<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MinP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_PartialDiv_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP4 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP5 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0AddP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP5 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0SubP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP5 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP5 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P5() {
    type A = Z0;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN5 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1AddN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN5 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1SubN5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MulN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<P1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN5 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Pow_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN4 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1AddN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN4 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1SubN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P1MulN4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MulN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN4 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1AddN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P1SubN3 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1SubN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MulN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN3 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N2() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1AddN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1SubN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MulN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1MaxN2 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1AddN1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1AddN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1SubN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MulN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivN1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1DivN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P1PartialDivN1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Add_0 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Add_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub__0() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Sub_0 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Sub_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Max_0 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Max_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P1Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1AddP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1SubP1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1SubP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Mul_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MulP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1DivP1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1DivP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P1PartialDivP1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP1 as Ord>::to_ordering(), Ordering::Equal);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1AddP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1SubP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MulP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP2 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P2() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP3 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1AddP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1SubP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MulP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Min_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP3 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP4 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1AddP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP4 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1SubP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MulP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP4 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P4() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP5 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P1AddP5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP5 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1SubP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MulP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP5 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Gcd_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN5 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2AddN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN5 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2SubN5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P2MulN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN5 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddN4 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2AddN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN4 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2SubN4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P2MulN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<P2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN4 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2AddN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN3 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2SubN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P2MulN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN3 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2AddN2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2AddN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN2 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2SubN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MulN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P2PartialDivN2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2AddN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2SubN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MulN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P2DivN1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2DivN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P2PartialDivN1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Add_0 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Add_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Sub_0 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Sub_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Max_0 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Max_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P2Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2AddP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2SubP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MulP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivP1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2DivP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P2PartialDivP1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP1 = <<A as Pow<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2PowP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP1 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<P2CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP2 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2AddP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2SubP2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2SubP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MulP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P2PartialDivP2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP3 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2AddP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2SubP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2MulP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP3 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP3 = <<A as Pow<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2PowP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP4 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P2AddP4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubP4 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2SubP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2MulP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP4 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP5 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2AddP5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP5 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2SubP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P2MulP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP5 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P32 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP5 = <<A as Pow<B>>::Output as Same<P32>>::Output;

    assert_eq!(<P2PowP5 as Integer>::to_i64(), <P32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN5 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3AddN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN5 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3SubN5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN5 = <<A as Mul<B>>::Output as Same<N15>>::Output;
```

```
    assert_eq!(<P3MulN5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN5 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3AddN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN4 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3SubN4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P3MulN4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P3MulN4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN4 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3AddN3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3AddN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN3 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3SubN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3MulN3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<P3MulN3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P3DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P3PartialDivN3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3AddN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN2 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3SubN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P3MulN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3AddN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3SubN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MulN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3DivN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<P3PartialDivN1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Add_0 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Add_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Sub_0 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Sub_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Max_0 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Max_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P3Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP1 = <<A as Add<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P3AddP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3SubP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MulP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3DivP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<P3PartialDivP1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3PowP1 = <<A as Pow<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3PowP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP2 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3AddP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3SubP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3MulP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP3 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3AddP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3SubP3 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P3SubP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3MulP3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P3PartialDivP3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P27 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP3 = <<A as Pow<B>>::Output as Same<P27>>::Output;

    assert_eq!(<P3PowP3 as Integer>::to_i64(), <P27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP4 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3AddP4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3SubP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P3MulP4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP4 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<P3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP5 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3AddP5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP5 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3SubP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP5 = <<A as Mul<B>>::Output as Same<P15>>::Output;
```

```
    assert_eq!(<P3MulP5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP5 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P243 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type P3PowP5 = <<A as Pow<B>>::Output as Same<P243>>::Output;

    assert_eq!(<P3PowP5 as Integer>::to_i64(), <P243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN5 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4AddN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P4SubN5 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4SubN5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P4MulN5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN5 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemN5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4AddN4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4AddN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4SubN4 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4SubN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<P4MulN4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P4PartialDivN4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4AddN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN3 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4SubN3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P4MulN3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P4MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P4MaxN3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddN2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4AddN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN2 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4SubN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P4MulN2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4MaxN2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu
```

```
    assert_eq!(<P4PartialDivN2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4AddN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN1 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4SubN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MulN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4DivN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type P4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<P4PartialDivN1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Add_0 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Add_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Sub_0 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Sub_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Max_0 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Max_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P4Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP1 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4AddP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4SubP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MulP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4DivP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<P4PartialDivP1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PowP1 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4PowP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP2 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P4AddP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubP2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4SubP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4MulP2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P4PartialDivP2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type P4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP3 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4AddP3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4SubP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P4MulP3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P64 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP3 = <<A as Pow<B>>::Output as Same<P64>>::Output;

    assert_eq!(<P4PowP3 as Integer>::to_i64(), <P64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP4 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4AddP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4SubP4 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P4SubP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4MulP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P4PartialDivP4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type P4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<P4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP5 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4AddP5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP5 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4SubP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P4MulP5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP5 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemP5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1024 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P4PowP5 = <<A as Pow<B>>::Output as Same<P1024>>::Output;

    assert_eq!(<P4PowP5 as Integer>::to_i64(), <P1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5AddN5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5AddN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN5 = <<A as Sub<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5SubN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulN5 = <<A as Mul<B>>::Output as Same<N25>>::Output;
```

```
    assert_eq!(<P5MulN5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P5PartialDivN5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5AddN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5SubN4 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5SubN4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulN4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P5MulN4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P5MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5AddN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P5SubN3 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5SubN3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P5MulN3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P5MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5AddN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN2 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5SubN2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulN2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P5MulN2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5AddN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN1 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5SubN1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MulN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5DivN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<P5PartialDivN1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Add_0 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Add_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Sub_0 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Sub_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Max_0 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Max_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P5Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP1 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P5AddP1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5SubP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MulP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5DivP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<P5PartialDivP1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5PowP1 = <<A as Pow<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5PowP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP2 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5AddP2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5SubP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulP2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5MulP2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP3 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5AddP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP3 = <<A as Sub<B>>::Output as Same<P2>>::Output;
```

```
    assert_eq!(<P5SubP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P5MulP3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1
```

```
    #[allow(non_camel_case_types)]
    type P5PowP3 = <<A as Pow<B>>::Output as Same<P125>>::Output;

    assert_eq!(<P5PowP3 as Integer>::to_i64(), <P125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5AddP4 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5AddP4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5SubP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulP4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P5MulP4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type P5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<P5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP5 = <<A as Add<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5AddP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5SubP5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5SubP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulP5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5MulP5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP5 = <<A as Min<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P5MinP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_PartialDiv_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P5PartialDivP5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P5PowP5 = <<A as Pow<B>>::Output as Same<P3125>>::Output;

    assert_eq!(<P5PowP5 as Integer>::to_i64(), <P3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN5 = <<A as Neg>::Output as Same<P5>>::Output;
    assert_eq!(<NegN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN4 = <<A as Neg>::Output as Same<P4>>::Output;
    assert_eq!(<NegN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN3 = <<A as Neg>::Output as Same<P3>>::Output;
    assert_eq!(<NegN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN2 = <<A as Neg>::Output as Same<P2>>::Output;
    assert_eq!(<NegN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Neg() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegN1 = <<A as Neg>::Output as Same<P1>>::Output;
    assert_eq!(<NegN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Abs() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Neg() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Neg_0 = <<A as Neg>::Output as Same<_0>>::Output;
    assert_eq!(<Neg_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Abs() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Abs_0 = <<A as Abs>::Output as Same<_0>>::Output;
    assert_eq!(<Abs_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Neg() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegP1 = <<A as Neg>::Output as Same<N1>>::Output;
    assert_eq!(<NegP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Abs() {
    type A = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP2 = <<A as Neg>::Output as Same<N2>>::Output;
    assert_eq!(<NegP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP3 = <<A as Neg>::Output as Same<N3>>::Output;
    assert_eq!(<NegP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type AbsP3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP4 = <<A as Neg>::Output as Same<N4>>::Output;
    assert_eq!(<NegP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP5 = <<A as Neg>::Output as Same<N5>>::Output;
    assert_eq!(<NegP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

File: ./target/release/build/typenum-346c6668244bcd95/out/op.rs

```
/**
Convenient type operations.
```

Any types representing values must be able to be expressed as `ident`s. Tha
in scope.

For example, `P5` is okay, but `typenum::P5` is not.

You may combine operators arbitrarily, although doing so excessively may re
recursion limit.

Example
```rust
#![recursion_limit="128"]
#[macro_use] extern crate typenum;
use typenum::consts::*;

fn main() {
    assert_type!(
        op!(min((P1 - P2) * (N3 + N7), P5 * (P3 + P4)) == P10)
    );
}
```
Operators are evaluated based on the operator precedence outlined
[here](https://doc.rust-lang.org/reference.html#operator-precedence).

The full list of supported operators and functions is as follows:

`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, `==`, `!=`, `<=`, `>=`,

They all expand to type aliases defined in the `operator_aliases` module. H
including examples:

Operator `*`. Expands to `Prod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 * P3), P6);
# }
```

Operator `/`. Expands to `Quot`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 / P2), P3);
# }
```

Operator `%`. Expands to `Mod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 % P3), P2);
# }
```

Operator `+`. Expands to `Sum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 + P3), P5);
# }
```

Operator `-`. Expands to `Diff`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 - P3), N1);
# }
```

Operator `<<`. Expands to `Shleft`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U1 << U5), U32);
# }
```

Operator `>>`. Expands to `Shright`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
```

```
assert_type_eq!(op!(U32 >> U5), U1);
# }
```

Operator `&`. Expands to `And`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 & U3), U1);
# }
```

Operator `^`. Expands to `Xor`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 ^ U3), U6);
# }
```

Operator `|`. Expands to `Or`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 | U3), U7);
# }
```

Operator `==`. Expands to `Eq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 == P3 + P2), True);
# }
```

Operator `!=`. Expands to `NotEq`.

```rust
```

```
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 != P3 + P2), False);
# }
```

Operator `<=`. Expands to `LeEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 <= P3 + P2), False);
# }
```

Operator `>=`. Expands to `GrEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 >= P3 + P2), True);
# }
```

Operator `<`. Expands to `Le`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P4 < P3 + P2), True);
# }
```

Operator `>`. Expands to `Gr`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 < P3 + P2), False);
# }
```

Operator `cmp`. Expands to `Compare`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cmp(P2, P3)), Less);
# }
```

Operator `sqr`. Expands to `Square`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqr(P2)), P4);
# }
```

Operator `sqrt`. Expands to `Sqrt`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqrt(U9)), U3);
# }
```

Operator `abs`. Expands to `AbsVal`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(abs(N2)), P2);
# }
```

Operator `cube`. Expands to `Cube`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cube(P2)), P8);
# }
```

```

---
Operator `pow`. Expands to `Exp`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(pow(P2, P3)), P8);
# }
```

Operator `min`. Expands to `Minimum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(min(P2, P3)), P2);
# }
```

Operator `max`. Expands to `Maximum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(max(P2, P3)), P3);
# }
```

Operator `log2`. Expands to `Log2`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(log2(U9)), U3);
# }
```

Operator `gcd`. Expands to `Gcf`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
```
```

```
# fn main() {
assert_type_eq!(op!(gcd(U9, U21)), U3);
# }
```

*/
#[macro_export(local_inner_macros)]
macro_rules! op {
    ($($tail:tt)*) => (__op_internal__!($($tail)*));
}

    #[doc(hidden)]
    #[macro_export(local_inner_macros)]
    macro_rules! __op_internal__ {

(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cmp $($tail:tt)
    __op_internal__!(@stack[Compare, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqr $($tail:tt)
    __op_internal__!(@stack[Square, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqrt $($tail:tt
    __op_internal__!(@stack[Sqrt, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: abs $($tail:tt)
    __op_internal__!(@stack[AbsVal, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cube $($tail:tt
    __op_internal__!(@stack[Cube, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: pow $($tail:tt)
    __op_internal__!(@stack[Exp, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: min $($tail:tt)
    __op_internal__!(@stack[Minimum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: max $($tail:tt)
    __op_internal__!(@stack[Maximum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: log2 $($tail:tt
    __op_internal__!(@stack[Log2, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: gcd $($tail:tt)
    __op_internal__!(@stack[Gcf, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: , $($ta
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail

```
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: *
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: *
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: *
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:tt)*)
    __op_internal__!(@stack[Prod, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: /
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: /
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: /
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:tt)*)
    __op_internal__!(@stack[Quot, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: %
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: %
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: %
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:tt)*)
    __op_internal__!(@stack[Mod, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: +
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: +
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: +
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: +
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: +
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:tt)*)
    __op_internal__!(@stack[Sum, $($stack,)*] @queue[$($queue,)*] @tail: $(
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: -
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: -
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: -
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: -
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: -
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:tt)*)
    __op_internal__!(@stack[Diff, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <<
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <<
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail:tt)*
    __op_internal__!(@stack[Shleft, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >>
);
```

```
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >>
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail:tt)*
    __op_internal__!(@stack[Shright, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: &
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: &
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: &
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: &
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: &
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: &
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:tt)*)
    __op_internal__!(@stack[And, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: ^
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: ^
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ^
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ^
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: ^
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ^
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ^
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:tt)*)
    __op_internal__!(@stack[Xor, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: |
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: |
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: |
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: |
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: |
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: |
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: |
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: | $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:tt)*)
    __op_internal__!(@stack[Or, $($stack,)*] @queue[$($queue,)*] @tail: $($
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: =
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: =
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ==
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ==
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: =
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ==
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ==
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: ==
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: ==
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: =
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: =
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: ==
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: ==
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:tt)*
    __op_internal__!(@stack[Eq, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
```

```
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: !
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: !
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: !=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: !=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: !
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: !=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: !=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: !=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: !=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: !
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: !
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: !=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: !=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:tt)*
    __op_internal__!(@stack[NotEq, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: <=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: <=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: <=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: <=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:tt)*
    __op_internal__!(@stack[LeEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
```

```
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: >=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: >=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: >=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: >=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:tt)*
    __op_internal__!(@stack[GrEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
```

```
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: < $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: < $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: < $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: < $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:tt)*)
    __op_internal__!(@stack[Le, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: > $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: > $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: > $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: > $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:tt)*)
    __op_internal__!(@stack[Gr, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ( $($stuff:tt)*
 => (
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*]
                     @tail: $($stuff)* RParen $($tail)*)
);
```

```
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: RParen
    __op_internal__!(@rp3 @stack[$($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
 => (
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@rp3 @stack[Compare, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Compare, $($queue,)*] @tail
);
(@rp3 @stack[Square, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[Square, $($queue,)*] @tail:
);
(@rp3 @stack[Sqrt, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Sqrt, $($queue,)*] @tail: $
);
(@rp3 @stack[AbsVal, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[AbsVal, $($queue,)*] @tail:
);
(@rp3 @stack[Cube, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Cube, $($queue,)*] @tail: $
);
(@rp3 @stack[Exp, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Exp, $($queue,)*] @tail: $(
);
(@rp3 @stack[Minimum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Minimum, $($queue,)*] @tail
);
(@rp3 @stack[Maximum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Maximum, $($queue,)*] @tail
);
(@rp3 @stack[Log2, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Log2, $($queue,)*] @tail: $
);
(@rp3 @stack[Gcf, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Gcf, $($queue,)*] @tail: $(
);
(@rp3 @stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($tail:tt
    __op_internal__!(@stack[$($stack,)*] @queue[$($queue,)*] @tail: $($tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $num:ident $($t
    __op_internal__!(@stack[$($stack,)*] @queue[$num, $($queue,)*] @tail: $
);
(@stack[] @queue[$($queue:ident,)*] @tail: ) => (
    __op_internal__!(@reverse[] @input: $($queue,)*)
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@reverse[$($revved:ident,)*] @input: $head:ident, $($tail:ident,)* ) => (
    __op_internal__!(@reverse[$head, $($revved,)*] @input: $($tail,)*)
);
```

```
(@reverse[$($revved:ident,)*] @input: ) => (
    __op_internal__!(@eval @stack[] @input[$($revved,)*])
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Prod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Prod<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Quot, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Quot<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Mod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Mod<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Sum, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Sum<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Diff, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Diff<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shleft, $($tail:ident,)*
    __op_internal__!(@eval @stack[$crate::Shleft<$b, $a>, $($stack,)*] @inp
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shright, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Shright<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[And, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::And<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Xor, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Xor<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Or, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Or<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Eq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Eq<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[NotEq, $($tail:ident,)*]
    __op_internal__!(@eval @stack[$crate::NotEq<$b, $a>, $($stack,)*] @inpu
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[LeEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::LeEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[GrEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::GrEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Le, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Le<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gr, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Gr<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Compare, $($tail:ident,)
```

```
    __op_internal__!(@eval @stack[$crate::Compare<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Exp, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Exp<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Minimum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Minimum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Maximum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Maximum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gcf, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Gcf<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Square, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Square<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Sqrt, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Sqrt<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[AbsVal, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::AbsVal<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Cube, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Cube<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Log2, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Log2<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$($stack:ty,)*] @input[$head:ident, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$head, $($stack,)*] @input[$($tail,)*])
);
(@eval @stack[$stack:ty,] @input[]) => (
    $stack
);
($($tail:tt)* ) => (
    __op_internal__!(@stack[] @queue[] @tail: $($tail)*)
);
}
```

File: ./target/release/build/libsqlite3-sys-ef39d2d0e4291b45/out/bind

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
```

```
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
```

```
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
```

```
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
```

```
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
```

```
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
```

```
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
```

```
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
```

```
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
```

```
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
```

```
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
```

```
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xWrite: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *const ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
    >,
    pub xSync: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFileSize: ::std::option::Option<
```

```
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xDeviceCharacteristics: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xShmMap: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        iPg: ::std::os::raw::c_int,
        pgsz: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xShmLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        offset: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
```

```
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
    pub pNext: *mut sqlite3_vfs,
    pub zName: *const ::std::os::raw::c_char,
    pub pAppData: *mut ::std::os::raw::c_void,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: sqlite3_filename,
            arg2: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
            pOutFlags: *mut ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int,
>,
pub xDelete: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        syncDir: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xAccess: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        flags: ::std::os::raw::c_int,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFullPathname: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        nOut: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xDlOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zFilename: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
    >,
    pub xRandomness: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            nByte: ::std::os::raw::c_int,
            zOut: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSleep: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            microseconds: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCurrentTime: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
    >,
    pub xGetLastError: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: ::std::os::raw::c_int,
            arg3: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCurrentTimeInt64: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            arg2: sqlite3_syscall_ptr,
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
```

```
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
```

```
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
```

```
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
```

```
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
```

```
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
```

```
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
```

```
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
```

```
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
```

```
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
```

```
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
```

```
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
```

```
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
```

```
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
```

```
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
```

```
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
```

```
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
```

```
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
```

```
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
```

```
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
```

```
    ) -> ::std::os::raw::c_int,
>,
pub xConnect: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3,
        pAux: *mut ::std::os::raw::c_void,
        argc: ::std::os::raw::c_int,
        argv: *const *const ::std::os::raw::c_char,
        ppVTab: *mut *mut sqlite3_vtab,
        arg2: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xBestIndex: ::std::option::Option<
    unsafe extern "C" fn(
        pVTab: *mut sqlite3_vtab,
        arg1: *mut sqlite3_index_info,
    ) -> ::std::os::raw::c_int,
>,
pub xDisconnect: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xDestroy: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xOpen: ::std::option::Option<
    unsafe extern "C" fn(
        pVTab: *mut sqlite3_vtab,
        ppCursor: *mut *mut sqlite3_vtab_cursor,
    ) -> ::std::os::raw::c_int,
>,
pub xClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xFilter: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        idxNum: ::std::os::raw::c_int,
        idxStr: *const ::std::os::raw::c_char,
        argc: ::std::os::raw::c_int,
        argv: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int,
>,
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
```

```
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        nArg: ::std::os::raw::c_int,
        zName: *const ::std::os::raw::c_char,
        pxFunc: *mut ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        ppArg: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xRename: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        zNew: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSavepoint: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
```

```
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
```

```
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
```

```
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
```

```
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
```

```
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
```

```
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
```

```
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
```

```
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
```

```
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
```

```
pub xUserData: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
>,
pub xColumnCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xRowCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnRow: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnTotalSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xTokenize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pText: *const ::std::os::raw::c_char,
        nText: ::std::os::raw::c_int,
        pCtx: *mut ::std::os::raw::c_void,
        xToken: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
                arg5: ::std::os::raw::c_int,
                arg6: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xPhraseCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xPhraseSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
```

```
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pAux: *mut ::std::os::raw::c_void,
        xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int,
>,
pub xGetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        bClear: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void,
```

```
    >,
    pub xPhraseFirst: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
            arg4: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNext: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
            piOff: *mut ::std::os::raw::c_int,
        ),
    >,
    pub xPhraseFirstColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNextColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
```

```
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./target/x86_64-pc-windows-gnu/release/build/typenum-10a9d0

````
/**
Type aliases for many constants.

This file is generated by typenum's build script.

For unsigned integers, the format is `U` followed by the number. We define

- Numbers 0 through 1024
- Powers of 2 below `u64::MAX`
- Powers of 10 below `u64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm};

# #[allow(dead_code)]
type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

For positive signed integers, the format is `P` followed by the number and
signed integers it is `N` followed by the number. For the signed integer ze
`Z0`. We define aliases for

- Numbers -1024 through 1024
- Powers of 2 between `i64::MIN` and `i64::MAX`
- Powers of 10 between `i64::MIN` and `i64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm, PInt, NInt};

# #[allow(dead_code)]
type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
# #[allow(dead_code)]
type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```


# Example
```rust
# #[allow(unused_imports)]
use typenum::{U0, U1, U2, U3, U4, U5, U6};
# #[allow(unused_imports)]
use typenum::{N3, N2, N1, Z0, P1, P2, P3};
# #[allow(unused_imports)]
use typenum::{U774, N17, N10000, P1024, P4096};
```
````

```
We also define the aliases `False` and `True` for `B0` and `B1`, respective
*/
#[allow(missing_docs)]
pub mod consts {
    use crate::uint::{UInt, UTerm};
    use crate::int::{PInt, NInt};

    pub use crate::bit::{B0, B1};
    pub use crate::int::Z0;

    pub type True = B1;
    pub type False = B0;
    pub type U0 = UTerm;
    pub type U1 = UInt<UTerm, B1>;
    pub type P1 = PInt<U1>; pub type N1 = NInt<U1>;
    pub type U2 = UInt<UInt<UTerm, B1>, B0>;
    pub type P2 = PInt<U2>; pub type N2 = NInt<U2>;
    pub type U3 = UInt<UInt<UTerm, B1>, B1>;
    pub type P3 = PInt<U3>; pub type N3 = NInt<U3>;
    pub type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    pub type P4 = PInt<U4>; pub type N4 = NInt<U4>;
    pub type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    pub type P5 = PInt<U5>; pub type N5 = NInt<U5>;
    pub type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
    pub type P6 = PInt<U6>; pub type N6 = NInt<U6>;
    pub type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
    pub type P7 = PInt<U7>; pub type N7 = NInt<U7>;
    pub type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
    pub type P8 = PInt<U8>; pub type N8 = NInt<U8>;
    pub type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;
    pub type P9 = PInt<U9>; pub type N9 = NInt<U9>;
    pub type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;
    pub type P10 = PInt<U10>; pub type N10 = NInt<U10>;
    pub type U11 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>;
    pub type P11 = PInt<U11>; pub type N11 = NInt<U11>;
    pub type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;
    pub type P12 = PInt<U12>; pub type N12 = NInt<U12>;
    pub type U13 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>;
    pub type P13 = PInt<U13>; pub type N13 = NInt<U13>;
    pub type U14 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>;
    pub type P14 = PInt<U14>; pub type N14 = NInt<U14>;
    pub type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;
    pub type P15 = PInt<U15>; pub type N15 = NInt<U15>;
    pub type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
    pub type P16 = PInt<U16>; pub type N16 = NInt<U16>;
    pub type U17 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>;
    pub type P17 = PInt<U17>; pub type N17 = NInt<U17>;
    pub type U18 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B0>;
    pub type P18 = PInt<U18>; pub type N18 = NInt<U18>;
    pub type U19 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B1>;
    pub type P19 = PInt<U19>; pub type N19 = NInt<U19>;
    pub type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;
```

```
pub type P20 = PInt<U20>; pub type N20 = NInt<U20>;
pub type U21 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B1>;
pub type P21 = PInt<U21>; pub type N21 = NInt<U21>;
pub type U22 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B0>;
pub type P22 = PInt<U22>; pub type N22 = NInt<U22>;
pub type U23 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B1>;
pub type P23 = PInt<U23>; pub type N23 = NInt<U23>;
pub type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;
pub type P24 = PInt<U24>; pub type N24 = NInt<U24>;
pub type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
pub type P25 = PInt<U25>; pub type N25 = NInt<U25>;
pub type U26 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B0>;
pub type P26 = PInt<U26>; pub type N26 = NInt<U26>;
pub type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;
pub type P27 = PInt<U27>; pub type N27 = NInt<U27>;
pub type U28 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B0>;
pub type P28 = PInt<U28>; pub type N28 = NInt<U28>;
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
pub type U35 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P35 = PInt<U35>; pub type N35 = NInt<U35>;
pub type U36 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P36 = PInt<U36>; pub type N36 = NInt<U36>;
pub type U37 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P37 = PInt<U37>; pub type N37 = NInt<U37>;
pub type U38 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P38 = PInt<U38>; pub type N38 = NInt<U38>;
pub type U39 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P39 = PInt<U39>; pub type N39 = NInt<U39>;
pub type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P40 = PInt<U40>; pub type N40 = NInt<U40>;
pub type U41 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P41 = PInt<U41>; pub type N41 = NInt<U41>;
pub type U42 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P42 = PInt<U42>; pub type N42 = NInt<U42>;
pub type U43 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P43 = PInt<U43>; pub type N43 = NInt<U43>;
pub type U44 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P44 = PInt<U44>; pub type N44 = NInt<U44>;
pub type U45 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P45 = PInt<U45>; pub type N45 = NInt<U45>;
pub type U46 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
```

```
pub type P46 = PInt<U46>; pub type N46 = NInt<U46>;
pub type U47 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P47 = PInt<U47>; pub type N47 = NInt<U47>;
pub type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P48 = PInt<U48>; pub type N48 = NInt<U48>;
pub type U49 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P49 = PInt<U49>; pub type N49 = NInt<U49>;
pub type U50 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P50 = PInt<U50>; pub type N50 = NInt<U50>;
pub type U51 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P51 = PInt<U51>; pub type N51 = NInt<U51>;
pub type U52 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P52 = PInt<U52>; pub type N52 = NInt<U52>;
pub type U53 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P53 = PInt<U53>; pub type N53 = NInt<U53>;
pub type U54 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P54 = PInt<U54>; pub type N54 = NInt<U54>;
pub type U55 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P55 = PInt<U55>; pub type N55 = NInt<U55>;
pub type U56 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P56 = PInt<U56>; pub type N56 = NInt<U56>;
pub type U57 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P57 = PInt<U57>; pub type N57 = NInt<U57>;
pub type U58 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P58 = PInt<U58>; pub type N58 = NInt<U58>;
pub type U59 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P59 = PInt<U59>; pub type N59 = NInt<U59>;
pub type U60 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P60 = PInt<U60>; pub type N60 = NInt<U60>;
pub type U61 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P61 = PInt<U61>; pub type N61 = NInt<U61>;
pub type U62 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P62 = PInt<U62>; pub type N62 = NInt<U62>;
pub type U63 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P63 = PInt<U63>; pub type N63 = NInt<U63>;
pub type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P64 = PInt<U64>; pub type N64 = NInt<U64>;
pub type U65 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P65 = PInt<U65>; pub type N65 = NInt<U65>;
pub type U66 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P66 = PInt<U66>; pub type N66 = NInt<U66>;
pub type U67 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P67 = PInt<U67>; pub type N67 = NInt<U67>;
pub type U68 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P68 = PInt<U68>; pub type N68 = NInt<U68>;
pub type U69 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P69 = PInt<U69>; pub type N69 = NInt<U69>;
pub type U70 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P70 = PInt<U70>; pub type N70 = NInt<U70>;
pub type U71 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P71 = PInt<U71>; pub type N71 = NInt<U71>;
pub type U72 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
```

```
pub type P72 = PInt<U72>; pub type N72 = NInt<U72>;
pub type U73 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P73 = PInt<U73>; pub type N73 = NInt<U73>;
pub type U74 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P74 = PInt<U74>; pub type N74 = NInt<U74>;
pub type U75 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P75 = PInt<U75>; pub type N75 = NInt<U75>;
pub type U76 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P76 = PInt<U76>; pub type N76 = NInt<U76>;
pub type U77 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P77 = PInt<U77>; pub type N77 = NInt<U77>;
pub type U78 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P78 = PInt<U78>; pub type N78 = NInt<U78>;
pub type U79 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P79 = PInt<U79>; pub type N79 = NInt<U79>;
pub type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P80 = PInt<U80>; pub type N80 = NInt<U80>;
pub type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P81 = PInt<U81>; pub type N81 = NInt<U81>;
pub type U82 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P82 = PInt<U82>; pub type N82 = NInt<U82>;
pub type U83 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P83 = PInt<U83>; pub type N83 = NInt<U83>;
pub type U84 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P84 = PInt<U84>; pub type N84 = NInt<U84>;
pub type U85 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P85 = PInt<U85>; pub type N85 = NInt<U85>;
pub type U86 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P86 = PInt<U86>; pub type N86 = NInt<U86>;
pub type U87 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P87 = PInt<U87>; pub type N87 = NInt<U87>;
pub type U88 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P88 = PInt<U88>; pub type N88 = NInt<U88>;
pub type U89 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P89 = PInt<U89>; pub type N89 = NInt<U89>;
pub type U90 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P90 = PInt<U90>; pub type N90 = NInt<U90>;
pub type U91 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P91 = PInt<U91>; pub type N91 = NInt<U91>;
pub type U92 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P92 = PInt<U92>; pub type N92 = NInt<U92>;
pub type U93 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P93 = PInt<U93>; pub type N93 = NInt<U93>;
pub type U94 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P94 = PInt<U94>; pub type N94 = NInt<U94>;
pub type U95 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P95 = PInt<U95>; pub type N95 = NInt<U95>;
pub type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P96 = PInt<U96>; pub type N96 = NInt<U96>;
pub type U97 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P97 = PInt<U97>; pub type N97 = NInt<U97>;
pub type U98 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
```

```
pub type P98 = PInt<U98>; pub type N98 = NInt<U98>;
pub type U99 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P99 = PInt<U99>; pub type N99 = NInt<U99>;
pub type U100 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P100 = PInt<U100>; pub type N100 = NInt<U100>;
pub type U101 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P101 = PInt<U101>; pub type N101 = NInt<U101>;
pub type U102 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P102 = PInt<U102>; pub type N102 = NInt<U102>;
pub type U103 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P103 = PInt<U103>; pub type N103 = NInt<U103>;
pub type U104 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P104 = PInt<U104>; pub type N104 = NInt<U104>;
pub type U105 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P105 = PInt<U105>; pub type N105 = NInt<U105>;
pub type U106 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P106 = PInt<U106>; pub type N106 = NInt<U106>;
pub type U107 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P107 = PInt<U107>; pub type N107 = NInt<U107>;
pub type U108 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P108 = PInt<U108>; pub type N108 = NInt<U108>;
pub type U109 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P109 = PInt<U109>; pub type N109 = NInt<U109>;
pub type U110 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P110 = PInt<U110>; pub type N110 = NInt<U110>;
pub type U111 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P111 = PInt<U111>; pub type N111 = NInt<U111>;
pub type U112 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P112 = PInt<U112>; pub type N112 = NInt<U112>;
pub type U113 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P113 = PInt<U113>; pub type N113 = NInt<U113>;
pub type U114 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P114 = PInt<U114>; pub type N114 = NInt<U114>;
pub type U115 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P115 = PInt<U115>; pub type N115 = NInt<U115>;
pub type U116 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P116 = PInt<U116>; pub type N116 = NInt<U116>;
pub type U117 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P117 = PInt<U117>; pub type N117 = NInt<U117>;
pub type U118 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P118 = PInt<U118>; pub type N118 = NInt<U118>;
pub type U119 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P119 = PInt<U119>; pub type N119 = NInt<U119>;
pub type U120 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P120 = PInt<U120>; pub type N120 = NInt<U120>;
pub type U121 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P121 = PInt<U121>; pub type N121 = NInt<U121>;
pub type U122 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P122 = PInt<U122>; pub type N122 = NInt<U122>;
pub type U123 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P123 = PInt<U123>; pub type N123 = NInt<U123>;
pub type U124 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
```

```
pub type P124 = PInt<U124>; pub type N124 = NInt<U124>;
pub type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P125 = PInt<U125>; pub type N125 = NInt<U125>;
pub type U126 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P126 = PInt<U126>; pub type N126 = NInt<U126>;
pub type U127 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P127 = PInt<U127>; pub type N127 = NInt<U127>;
pub type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P128 = PInt<U128>; pub type N128 = NInt<U128>;
pub type U129 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P129 = PInt<U129>; pub type N129 = NInt<U129>;
pub type U130 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P130 = PInt<U130>; pub type N130 = NInt<U130>;
pub type U131 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P131 = PInt<U131>; pub type N131 = NInt<U131>;
pub type U132 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P132 = PInt<U132>; pub type N132 = NInt<U132>;
pub type U133 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P133 = PInt<U133>; pub type N133 = NInt<U133>;
pub type U134 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P134 = PInt<U134>; pub type N134 = NInt<U134>;
pub type U135 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P135 = PInt<U135>; pub type N135 = NInt<U135>;
pub type U136 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P136 = PInt<U136>; pub type N136 = NInt<U136>;
pub type U137 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P137 = PInt<U137>; pub type N137 = NInt<U137>;
pub type U138 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P138 = PInt<U138>; pub type N138 = NInt<U138>;
pub type U139 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P139 = PInt<U139>; pub type N139 = NInt<U139>;
pub type U140 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P140 = PInt<U140>; pub type N140 = NInt<U140>;
pub type U141 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P141 = PInt<U141>; pub type N141 = NInt<U141>;
pub type U142 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P142 = PInt<U142>; pub type N142 = NInt<U142>;
pub type U143 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P143 = PInt<U143>; pub type N143 = NInt<U143>;
pub type U144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P144 = PInt<U144>; pub type N144 = NInt<U144>;
pub type U145 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P145 = PInt<U145>; pub type N145 = NInt<U145>;
pub type U146 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P146 = PInt<U146>; pub type N146 = NInt<U146>;
pub type U147 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P147 = PInt<U147>; pub type N147 = NInt<U147>;
pub type U148 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P148 = PInt<U148>; pub type N148 = NInt<U148>;
pub type U149 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P149 = PInt<U149>; pub type N149 = NInt<U149>;
pub type U150 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
```

```
pub type P150 = PInt<U150>; pub type N150 = NInt<U150>;
pub type U151 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P151 = PInt<U151>; pub type N151 = NInt<U151>;
pub type U152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P152 = PInt<U152>; pub type N152 = NInt<U152>;
pub type U153 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P153 = PInt<U153>; pub type N153 = NInt<U153>;
pub type U154 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P154 = PInt<U154>; pub type N154 = NInt<U154>;
pub type U155 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P155 = PInt<U155>; pub type N155 = NInt<U155>;
pub type U156 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P156 = PInt<U156>; pub type N156 = NInt<U156>;
pub type U157 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P157 = PInt<U157>; pub type N157 = NInt<U157>;
pub type U158 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P158 = PInt<U158>; pub type N158 = NInt<U158>;
pub type U159 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P159 = PInt<U159>; pub type N159 = NInt<U159>;
pub type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P160 = PInt<U160>; pub type N160 = NInt<U160>;
pub type U161 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P161 = PInt<U161>; pub type N161 = NInt<U161>;
pub type U162 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P162 = PInt<U162>; pub type N162 = NInt<U162>;
pub type U163 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P163 = PInt<U163>; pub type N163 = NInt<U163>;
pub type U164 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P164 = PInt<U164>; pub type N164 = NInt<U164>;
pub type U165 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P165 = PInt<U165>; pub type N165 = NInt<U165>;
pub type U166 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P166 = PInt<U166>; pub type N166 = NInt<U166>;
pub type U167 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P167 = PInt<U167>; pub type N167 = NInt<U167>;
pub type U168 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P168 = PInt<U168>; pub type N168 = NInt<U168>;
pub type U169 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P169 = PInt<U169>; pub type N169 = NInt<U169>;
pub type U170 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P170 = PInt<U170>; pub type N170 = NInt<U170>;
pub type U171 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P171 = PInt<U171>; pub type N171 = NInt<U171>;
pub type U172 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P172 = PInt<U172>; pub type N172 = NInt<U172>;
pub type U173 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P173 = PInt<U173>; pub type N173 = NInt<U173>;
pub type U174 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P174 = PInt<U174>; pub type N174 = NInt<U174>;
pub type U175 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P175 = PInt<U175>; pub type N175 = NInt<U175>;
pub type U176 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
```

```
pub type P176 = PInt<U176>; pub type N176 = NInt<U176>;
pub type U177 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P177 = PInt<U177>; pub type N177 = NInt<U177>;
pub type U178 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P178 = PInt<U178>; pub type N178 = NInt<U178>;
pub type U179 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P179 = PInt<U179>; pub type N179 = NInt<U179>;
pub type U180 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P180 = PInt<U180>; pub type N180 = NInt<U180>;
pub type U181 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P181 = PInt<U181>; pub type N181 = NInt<U181>;
pub type U182 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P182 = PInt<U182>; pub type N182 = NInt<U182>;
pub type U183 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P183 = PInt<U183>; pub type N183 = NInt<U183>;
pub type U184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P184 = PInt<U184>; pub type N184 = NInt<U184>;
pub type U185 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P185 = PInt<U185>; pub type N185 = NInt<U185>;
pub type U186 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P186 = PInt<U186>; pub type N186 = NInt<U186>;
pub type U187 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P187 = PInt<U187>; pub type N187 = NInt<U187>;
pub type U188 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P188 = PInt<U188>; pub type N188 = NInt<U188>;
pub type U189 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P189 = PInt<U189>; pub type N189 = NInt<U189>;
pub type U190 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P190 = PInt<U190>; pub type N190 = NInt<U190>;
pub type U191 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P191 = PInt<U191>; pub type N191 = NInt<U191>;
pub type U192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P192 = PInt<U192>; pub type N192 = NInt<U192>;
pub type U193 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P193 = PInt<U193>; pub type N193 = NInt<U193>;
pub type U194 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P194 = PInt<U194>; pub type N194 = NInt<U194>;
pub type U195 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P195 = PInt<U195>; pub type N195 = NInt<U195>;
pub type U196 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P196 = PInt<U196>; pub type N196 = NInt<U196>;
pub type U197 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P197 = PInt<U197>; pub type N197 = NInt<U197>;
pub type U198 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P198 = PInt<U198>; pub type N198 = NInt<U198>;
pub type U199 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P199 = PInt<U199>; pub type N199 = NInt<U199>;
pub type U200 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P200 = PInt<U200>; pub type N200 = NInt<U200>;
pub type U201 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P201 = PInt<U201>; pub type N201 = NInt<U201>;
pub type U202 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P202 = PInt<U202>; pub type N202 = NInt<U202>;
pub type U203 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P203 = PInt<U203>; pub type N203 = NInt<U203>;
pub type U204 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P204 = PInt<U204>; pub type N204 = NInt<U204>;
pub type U205 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P205 = PInt<U205>; pub type N205 = NInt<U205>;
pub type U206 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P206 = PInt<U206>; pub type N206 = NInt<U206>;
pub type U207 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P207 = PInt<U207>; pub type N207 = NInt<U207>;
pub type U208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P208 = PInt<U208>; pub type N208 = NInt<U208>;
pub type U209 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P209 = PInt<U209>; pub type N209 = NInt<U209>;
pub type U210 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P210 = PInt<U210>; pub type N210 = NInt<U210>;
pub type U211 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P211 = PInt<U211>; pub type N211 = NInt<U211>;
pub type U212 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P212 = PInt<U212>; pub type N212 = NInt<U212>;
pub type U213 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P213 = PInt<U213>; pub type N213 = NInt<U213>;
pub type U214 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P214 = PInt<U214>; pub type N214 = NInt<U214>;
pub type U215 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P215 = PInt<U215>; pub type N215 = NInt<U215>;
pub type U216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P216 = PInt<U216>; pub type N216 = NInt<U216>;
pub type U217 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P217 = PInt<U217>; pub type N217 = NInt<U217>;
pub type U218 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P218 = PInt<U218>; pub type N218 = NInt<U218>;
pub type U219 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P219 = PInt<U219>; pub type N219 = NInt<U219>;
pub type U220 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P220 = PInt<U220>; pub type N220 = NInt<U220>;
pub type U221 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P221 = PInt<U221>; pub type N221 = NInt<U221>;
pub type U222 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P222 = PInt<U222>; pub type N222 = NInt<U222>;
pub type U223 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P223 = PInt<U223>; pub type N223 = NInt<U223>;
pub type U224 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P224 = PInt<U224>; pub type N224 = NInt<U224>;
pub type U225 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P225 = PInt<U225>; pub type N225 = NInt<U225>;
pub type U226 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P226 = PInt<U226>; pub type N226 = NInt<U226>;
pub type U227 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P227 = PInt<U227>; pub type N227 = NInt<U227>;
pub type U228 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P228 = PInt<U228>; pub type N228 = NInt<U228>;
pub type U229 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P229 = PInt<U229>; pub type N229 = NInt<U229>;
pub type U230 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P230 = PInt<U230>; pub type N230 = NInt<U230>;
pub type U231 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P231 = PInt<U231>; pub type N231 = NInt<U231>;
pub type U232 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P232 = PInt<U232>; pub type N232 = NInt<U232>;
pub type U233 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P233 = PInt<U233>; pub type N233 = NInt<U233>;
pub type U234 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P234 = PInt<U234>; pub type N234 = NInt<U234>;
pub type U235 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P235 = PInt<U235>; pub type N235 = NInt<U235>;
pub type U236 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P236 = PInt<U236>; pub type N236 = NInt<U236>;
pub type U237 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P237 = PInt<U237>; pub type N237 = NInt<U237>;
pub type U238 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P238 = PInt<U238>; pub type N238 = NInt<U238>;
pub type U239 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P239 = PInt<U239>; pub type N239 = NInt<U239>;
pub type U240 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P240 = PInt<U240>; pub type N240 = NInt<U240>;
pub type U241 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P241 = PInt<U241>; pub type N241 = NInt<U241>;
pub type U242 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P242 = PInt<U242>; pub type N242 = NInt<U242>;
pub type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P243 = PInt<U243>; pub type N243 = NInt<U243>;
pub type U244 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P244 = PInt<U244>; pub type N244 = NInt<U244>;
pub type U245 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P245 = PInt<U245>; pub type N245 = NInt<U245>;
pub type U246 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P246 = PInt<U246>; pub type N246 = NInt<U246>;
pub type U247 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P247 = PInt<U247>; pub type N247 = NInt<U247>;
pub type U248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P248 = PInt<U248>; pub type N248 = NInt<U248>;
pub type U249 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P249 = PInt<U249>; pub type N249 = NInt<U249>;
pub type U250 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P250 = PInt<U250>; pub type N250 = NInt<U250>;
pub type U251 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P251 = PInt<U251>; pub type N251 = NInt<U251>;
pub type U252 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P252 = PInt<U252>; pub type N252 = NInt<U252>;
pub type U253 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P253 = PInt<U253>; pub type N253 = NInt<U253>;
pub type U254 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P254 = PInt<U254>; pub type N254 = NInt<U254>;
pub type U255 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P255 = PInt<U255>; pub type N255 = NInt<U255>;
pub type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P256 = PInt<U256>; pub type N256 = NInt<U256>;
pub type U257 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P257 = PInt<U257>; pub type N257 = NInt<U257>;
pub type U258 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P258 = PInt<U258>; pub type N258 = NInt<U258>;
pub type U259 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P259 = PInt<U259>; pub type N259 = NInt<U259>;
pub type U260 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P260 = PInt<U260>; pub type N260 = NInt<U260>;
pub type U261 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P261 = PInt<U261>; pub type N261 = NInt<U261>;
pub type U262 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P262 = PInt<U262>; pub type N262 = NInt<U262>;
pub type U263 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P263 = PInt<U263>; pub type N263 = NInt<U263>;
pub type U264 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P264 = PInt<U264>; pub type N264 = NInt<U264>;
pub type U265 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P265 = PInt<U265>; pub type N265 = NInt<U265>;
pub type U266 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P266 = PInt<U266>; pub type N266 = NInt<U266>;
pub type U267 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P267 = PInt<U267>; pub type N267 = NInt<U267>;
pub type U268 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P268 = PInt<U268>; pub type N268 = NInt<U268>;
pub type U269 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P269 = PInt<U269>; pub type N269 = NInt<U269>;
pub type U270 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P270 = PInt<U270>; pub type N270 = NInt<U270>;
pub type U271 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P271 = PInt<U271>; pub type N271 = NInt<U271>;
pub type U272 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P272 = PInt<U272>; pub type N272 = NInt<U272>;
pub type U273 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P273 = PInt<U273>; pub type N273 = NInt<U273>;
pub type U274 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P274 = PInt<U274>; pub type N274 = NInt<U274>;
pub type U275 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P275 = PInt<U275>; pub type N275 = NInt<U275>;
pub type U276 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P276 = PInt<U276>; pub type N276 = NInt<U276>;
pub type U277 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P277 = PInt<U277>; pub type N277 = NInt<U277>;
pub type U278 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P278 = PInt<U278>; pub type N278 = NInt<U278>;
pub type U279 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P279 = PInt<U279>; pub type N279 = NInt<U279>;
pub type U280 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P280 = PInt<U280>; pub type N280 = NInt<U280>;
pub type U281 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P281 = PInt<U281>; pub type N281 = NInt<U281>;
pub type U282 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P282 = PInt<U282>; pub type N282 = NInt<U282>;
pub type U283 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P283 = PInt<U283>; pub type N283 = NInt<U283>;
pub type U284 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P284 = PInt<U284>; pub type N284 = NInt<U284>;
pub type U285 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P285 = PInt<U285>; pub type N285 = NInt<U285>;
pub type U286 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P286 = PInt<U286>; pub type N286 = NInt<U286>;
pub type U287 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P287 = PInt<U287>; pub type N287 = NInt<U287>;
pub type U288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P288 = PInt<U288>; pub type N288 = NInt<U288>;
pub type U289 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P289 = PInt<U289>; pub type N289 = NInt<U289>;
pub type U290 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P290 = PInt<U290>; pub type N290 = NInt<U290>;
pub type U291 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P291 = PInt<U291>; pub type N291 = NInt<U291>;
pub type U292 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P292 = PInt<U292>; pub type N292 = NInt<U292>;
pub type U293 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P293 = PInt<U293>; pub type N293 = NInt<U293>;
pub type U294 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P294 = PInt<U294>; pub type N294 = NInt<U294>;
pub type U295 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P295 = PInt<U295>; pub type N295 = NInt<U295>;
pub type U296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P296 = PInt<U296>; pub type N296 = NInt<U296>;
pub type U297 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P297 = PInt<U297>; pub type N297 = NInt<U297>;
pub type U298 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P298 = PInt<U298>; pub type N298 = NInt<U298>;
pub type U299 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P299 = PInt<U299>; pub type N299 = NInt<U299>;
pub type U300 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P300 = PInt<U300>; pub type N300 = NInt<U300>;
pub type U301 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P301 = PInt<U301>; pub type N301 = NInt<U301>;
pub type U302 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P302 = PInt<U302>; pub type N302 = NInt<U302>;
pub type U303 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P303 = PInt<U303>; pub type N303 = NInt<U303>;
pub type U304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P304 = PInt<U304>; pub type N304 = NInt<U304>;
pub type U305 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P305 = PInt<U305>; pub type N305 = NInt<U305>;
pub type U306 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P306 = PInt<U306>; pub type N306 = NInt<U306>;
pub type U307 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P307 = PInt<U307>; pub type N307 = NInt<U307>;
pub type U308 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P308 = PInt<U308>; pub type N308 = NInt<U308>;
pub type U309 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P309 = PInt<U309>; pub type N309 = NInt<U309>;
pub type U310 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P310 = PInt<U310>; pub type N310 = NInt<U310>;
pub type U311 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P311 = PInt<U311>; pub type N311 = NInt<U311>;
pub type U312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P312 = PInt<U312>; pub type N312 = NInt<U312>;
pub type U313 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P313 = PInt<U313>; pub type N313 = NInt<U313>;
pub type U314 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P314 = PInt<U314>; pub type N314 = NInt<U314>;
pub type U315 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P315 = PInt<U315>; pub type N315 = NInt<U315>;
pub type U316 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P316 = PInt<U316>; pub type N316 = NInt<U316>;
pub type U317 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P317 = PInt<U317>; pub type N317 = NInt<U317>;
pub type U318 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P318 = PInt<U318>; pub type N318 = NInt<U318>;
pub type U319 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P319 = PInt<U319>; pub type N319 = NInt<U319>;
pub type U320 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P320 = PInt<U320>; pub type N320 = NInt<U320>;
pub type U321 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P321 = PInt<U321>; pub type N321 = NInt<U321>;
pub type U322 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P322 = PInt<U322>; pub type N322 = NInt<U322>;
pub type U323 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P323 = PInt<U323>; pub type N323 = NInt<U323>;
pub type U324 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P324 = PInt<U324>; pub type N324 = NInt<U324>;
pub type U325 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P325 = PInt<U325>; pub type N325 = NInt<U325>;
pub type U326 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P326 = PInt<U326>; pub type N326 = NInt<U326>;
pub type U327 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P327 = PInt<U327>; pub type N327 = NInt<U327>;
pub type U328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P328 = PInt<U328>; pub type N328 = NInt<U328>;
pub type U329 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P329 = PInt<U329>; pub type N329 = NInt<U329>;
pub type U330 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P330 = PInt<U330>; pub type N330 = NInt<U330>;
pub type U331 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P331 = PInt<U331>; pub type N331 = NInt<U331>;
pub type U332 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P332 = PInt<U332>; pub type N332 = NInt<U332>;
pub type U333 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P333 = PInt<U333>; pub type N333 = NInt<U333>;
pub type U334 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P334 = PInt<U334>; pub type N334 = NInt<U334>;
pub type U335 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P335 = PInt<U335>; pub type N335 = NInt<U335>;
pub type U336 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P336 = PInt<U336>; pub type N336 = NInt<U336>;
pub type U337 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P337 = PInt<U337>; pub type N337 = NInt<U337>;
pub type U338 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P338 = PInt<U338>; pub type N338 = NInt<U338>;
pub type U339 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P339 = PInt<U339>; pub type N339 = NInt<U339>;
pub type U340 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P340 = PInt<U340>; pub type N340 = NInt<U340>;
pub type U341 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P341 = PInt<U341>; pub type N341 = NInt<U341>;
pub type U342 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P342 = PInt<U342>; pub type N342 = NInt<U342>;
pub type U343 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P343 = PInt<U343>; pub type N343 = NInt<U343>;
pub type U344 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P344 = PInt<U344>; pub type N344 = NInt<U344>;
pub type U345 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P345 = PInt<U345>; pub type N345 = NInt<U345>;
pub type U346 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P346 = PInt<U346>; pub type N346 = NInt<U346>;
pub type U347 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P347 = PInt<U347>; pub type N347 = NInt<U347>;
pub type U348 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P348 = PInt<U348>; pub type N348 = NInt<U348>;
pub type U349 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P349 = PInt<U349>; pub type N349 = NInt<U349>;
pub type U350 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P350 = PInt<U350>; pub type N350 = NInt<U350>;
pub type U351 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P351 = PInt<U351>; pub type N351 = NInt<U351>;
pub type U352 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P352 = PInt<U352>; pub type N352 = NInt<U352>;
pub type U353 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P353 = PInt<U353>; pub type N353 = NInt<U353>;
pub type U354 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P354 = PInt<U354>; pub type N354 = NInt<U354>;
pub type U355 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P355 = PInt<U355>; pub type N355 = NInt<U355>;
pub type U356 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P356 = PInt<U356>; pub type N356 = NInt<U356>;
pub type U357 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P357 = PInt<U357>; pub type N357 = NInt<U357>;
pub type U358 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P358 = PInt<U358>; pub type N358 = NInt<U358>;
pub type U359 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P359 = PInt<U359>; pub type N359 = NInt<U359>;
pub type U360 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P360 = PInt<U360>; pub type N360 = NInt<U360>;
pub type U361 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P361 = PInt<U361>; pub type N361 = NInt<U361>;
pub type U362 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P362 = PInt<U362>; pub type N362 = NInt<U362>;
pub type U363 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P363 = PInt<U363>; pub type N363 = NInt<U363>;
pub type U364 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P364 = PInt<U364>; pub type N364 = NInt<U364>;
pub type U365 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P365 = PInt<U365>; pub type N365 = NInt<U365>;
pub type U366 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P366 = PInt<U366>; pub type N366 = NInt<U366>;
pub type U367 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P367 = PInt<U367>; pub type N367 = NInt<U367>;
pub type U368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P368 = PInt<U368>; pub type N368 = NInt<U368>;
pub type U369 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P369 = PInt<U369>; pub type N369 = NInt<U369>;
pub type U370 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P370 = PInt<U370>; pub type N370 = NInt<U370>;
pub type U371 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P371 = PInt<U371>; pub type N371 = NInt<U371>;
pub type U372 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P372 = PInt<U372>; pub type N372 = NInt<U372>;
pub type U373 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P373 = PInt<U373>; pub type N373 = NInt<U373>;
pub type U374 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P374 = PInt<U374>; pub type N374 = NInt<U374>;
pub type U375 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P375 = PInt<U375>; pub type N375 = NInt<U375>;
pub type U376 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P376 = PInt<U376>; pub type N376 = NInt<U376>;
pub type U377 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P377 = PInt<U377>; pub type N377 = NInt<U377>;
pub type U378 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P378 = PInt<U378>; pub type N378 = NInt<U378>;
pub type U379 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P379 = PInt<U379>; pub type N379 = NInt<U379>;
pub type U380 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P380 = PInt<U380>; pub type N380 = NInt<U380>;
pub type U381 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P381 = PInt<U381>; pub type N381 = NInt<U381>;
pub type U382 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P382 = PInt<U382>; pub type N382 = NInt<U382>;
pub type U383 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P383 = PInt<U383>; pub type N383 = NInt<U383>;
pub type U384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P384 = PInt<U384>; pub type N384 = NInt<U384>;
pub type U385 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P385 = PInt<U385>; pub type N385 = NInt<U385>;
pub type U386 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P386 = PInt<U386>; pub type N386 = NInt<U386>;
pub type U387 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P387 = PInt<U387>; pub type N387 = NInt<U387>;
pub type U388 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P388 = PInt<U388>; pub type N388 = NInt<U388>;
pub type U389 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P389 = PInt<U389>; pub type N389 = NInt<U389>;
pub type U390 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P390 = PInt<U390>; pub type N390 = NInt<U390>;
pub type U391 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P391 = PInt<U391>; pub type N391 = NInt<U391>;
pub type U392 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P392 = PInt<U392>; pub type N392 = NInt<U392>;
pub type U393 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P393 = PInt<U393>; pub type N393 = NInt<U393>;
pub type U394 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P394 = PInt<U394>; pub type N394 = NInt<U394>;
pub type U395 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P395 = PInt<U395>; pub type N395 = NInt<U395>;
pub type U396 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P396 = PInt<U396>; pub type N396 = NInt<U396>;
pub type U397 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P397 = PInt<U397>; pub type N397 = NInt<U397>;
pub type U398 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P398 = PInt<U398>; pub type N398 = NInt<U398>;
pub type U399 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P399 = PInt<U399>; pub type N399 = NInt<U399>;
pub type U400 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P400 = PInt<U400>; pub type N400 = NInt<U400>;
pub type U401 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P401 = PInt<U401>; pub type N401 = NInt<U401>;
pub type U402 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P402 = PInt<U402>; pub type N402 = NInt<U402>;
pub type U403 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P403 = PInt<U403>; pub type N403 = NInt<U403>;
pub type U404 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P404 = PInt<U404>; pub type N404 = NInt<U404>;
pub type U405 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P405 = PInt<U405>; pub type N405 = NInt<U405>;
pub type U406 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P406 = PInt<U406>; pub type N406 = NInt<U406>;
pub type U407 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P407 = PInt<U407>; pub type N407 = NInt<U407>;
pub type U408 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P408 = PInt<U408>; pub type N408 = NInt<U408>;
pub type U409 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P409 = PInt<U409>; pub type N409 = NInt<U409>;
pub type U410 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P410 = PInt<U410>; pub type N410 = NInt<U410>;
pub type U411 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P411 = PInt<U411>; pub type N411 = NInt<U411>;
pub type U412 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P412 = PInt<U412>; pub type N412 = NInt<U412>;
pub type U413 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P413 = PInt<U413>; pub type N413 = NInt<U413>;
pub type U414 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P414 = PInt<U414>; pub type N414 = NInt<U414>;
pub type U415 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P415 = PInt<U415>; pub type N415 = NInt<U415>;
pub type U416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P416 = PInt<U416>; pub type N416 = NInt<U416>;
pub type U417 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P417 = PInt<U417>; pub type N417 = NInt<U417>;
pub type U418 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P418 = PInt<U418>; pub type N418 = NInt<U418>;
pub type U419 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P419 = PInt<U419>; pub type N419 = NInt<U419>;
pub type U420 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P420 = PInt<U420>; pub type N420 = NInt<U420>;
pub type U421 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P421 = PInt<U421>; pub type N421 = NInt<U421>;
pub type U422 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P422 = PInt<U422>; pub type N422 = NInt<U422>;
pub type U423 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P423 = PInt<U423>; pub type N423 = NInt<U423>;
pub type U424 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P424 = PInt<U424>; pub type N424 = NInt<U424>;
pub type U425 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P425 = PInt<U425>; pub type N425 = NInt<U425>;
pub type U426 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P426 = PInt<U426>; pub type N426 = NInt<U426>;
pub type U427 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P427 = PInt<U427>; pub type N427 = NInt<U427>;
pub type U428 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P428 = PInt<U428>; pub type N428 = NInt<U428>;
pub type U429 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P429 = PInt<U429>; pub type N429 = NInt<U429>;
pub type U430 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P430 = PInt<U430>; pub type N430 = NInt<U430>;
pub type U431 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P431 = PInt<U431>; pub type N431 = NInt<U431>;
pub type U432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P432 = PInt<U432>; pub type N432 = NInt<U432>;
pub type U433 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P433 = PInt<U433>; pub type N433 = NInt<U433>;
pub type U434 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P434 = PInt<U434>; pub type N434 = NInt<U434>;
pub type U435 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P435 = PInt<U435>; pub type N435 = NInt<U435>;
pub type U436 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P436 = PInt<U436>; pub type N436 = NInt<U436>;
pub type U437 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P437 = PInt<U437>; pub type N437 = NInt<U437>;
pub type U438 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P438 = PInt<U438>; pub type N438 = NInt<U438>;
pub type U439 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P439 = PInt<U439>; pub type N439 = NInt<U439>;
pub type U440 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P440 = PInt<U440>; pub type N440 = NInt<U440>;
pub type U441 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P441 = PInt<U441>; pub type N441 = NInt<U441>;
pub type U442 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P442 = PInt<U442>; pub type N442 = NInt<U442>;
pub type U443 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P443 = PInt<U443>; pub type N443 = NInt<U443>;
pub type U444 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P444 = PInt<U444>; pub type N444 = NInt<U444>;
pub type U445 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P445 = PInt<U445>; pub type N445 = NInt<U445>;
pub type U446 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P446 = PInt<U446>; pub type N446 = NInt<U446>;
pub type U447 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P447 = PInt<U447>; pub type N447 = NInt<U447>;
pub type U448 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P448 = PInt<U448>; pub type N448 = NInt<U448>;
pub type U449 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P449 = PInt<U449>; pub type N449 = NInt<U449>;
pub type U450 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P450 = PInt<U450>; pub type N450 = NInt<U450>;
pub type U451 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P451 = PInt<U451>; pub type N451 = NInt<U451>;
pub type U452 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P452 = PInt<U452>; pub type N452 = NInt<U452>;
pub type U453 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P453 = PInt<U453>; pub type N453 = NInt<U453>;
pub type U454 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P454 = PInt<U454>; pub type N454 = NInt<U454>;
pub type U455 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P455 = PInt<U455>; pub type N455 = NInt<U455>;
pub type U456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P456 = PInt<U456>; pub type N456 = NInt<U456>;
pub type U457 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P457 = PInt<U457>; pub type N457 = NInt<U457>;
pub type U458 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P458 = PInt<U458>; pub type N458 = NInt<U458>;
pub type U459 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P459 = PInt<U459>; pub type N459 = NInt<U459>;
pub type U460 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P460 = PInt<U460>; pub type N460 = NInt<U460>;
pub type U461 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P461 = PInt<U461>; pub type N461 = NInt<U461>;
pub type U462 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P462 = PInt<U462>; pub type N462 = NInt<U462>;
pub type U463 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P463 = PInt<U463>; pub type N463 = NInt<U463>;
pub type U464 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P464 = PInt<U464>; pub type N464 = NInt<U464>;
pub type U465 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P465 = PInt<U465>; pub type N465 = NInt<U465>;
pub type U466 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P466 = PInt<U466>; pub type N466 = NInt<U466>;
pub type U467 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P467 = PInt<U467>; pub type N467 = NInt<U467>;
pub type U468 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P468 = PInt<U468>; pub type N468 = NInt<U468>;
pub type U469 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P469 = PInt<U469>; pub type N469 = NInt<U469>;
pub type U470 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P470 = PInt<U470>; pub type N470 = NInt<U470>;
pub type U471 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P471 = PInt<U471>; pub type N471 = NInt<U471>;
pub type U472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P472 = PInt<U472>; pub type N472 = NInt<U472>;
pub type U473 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P473 = PInt<U473>; pub type N473 = NInt<U473>;
pub type U474 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P474 = PInt<U474>; pub type N474 = NInt<U474>;
pub type U475 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P475 = PInt<U475>; pub type N475 = NInt<U475>;
pub type U476 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P476 = PInt<U476>; pub type N476 = NInt<U476>;
pub type U477 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P477 = PInt<U477>; pub type N477 = NInt<U477>;
pub type U478 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P478 = PInt<U478>; pub type N478 = NInt<U478>;
pub type U479 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P479 = PInt<U479>; pub type N479 = NInt<U479>;
pub type U480 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P480 = PInt<U480>; pub type N480 = NInt<U480>;
pub type U481 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P481 = PInt<U481>; pub type N481 = NInt<U481>;
pub type U482 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P482 = PInt<U482>; pub type N482 = NInt<U482>;
pub type U483 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P483 = PInt<U483>; pub type N483 = NInt<U483>;
pub type U484 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P484 = PInt<U484>; pub type N484 = NInt<U484>;
pub type U485 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P485 = PInt<U485>; pub type N485 = NInt<U485>;
pub type U486 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P486 = PInt<U486>; pub type N486 = NInt<U486>;
pub type U487 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P487 = PInt<U487>; pub type N487 = NInt<U487>;
pub type U488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P488 = PInt<U488>; pub type N488 = NInt<U488>;
pub type U489 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P489 = PInt<U489>; pub type N489 = NInt<U489>;
pub type U490 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P490 = PInt<U490>; pub type N490 = NInt<U490>;
pub type U491 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P491 = PInt<U491>; pub type N491 = NInt<U491>;
pub type U492 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P492 = PInt<U492>; pub type N492 = NInt<U492>;
pub type U493 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P493 = PInt<U493>; pub type N493 = NInt<U493>;
pub type U494 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P494 = PInt<U494>; pub type N494 = NInt<U494>;
pub type U495 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P495 = PInt<U495>; pub type N495 = NInt<U495>;
pub type U496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P496 = PInt<U496>; pub type N496 = NInt<U496>;
pub type U497 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P497 = PInt<U497>; pub type N497 = NInt<U497>;
pub type U498 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P498 = PInt<U498>; pub type N498 = NInt<U498>;
pub type U499 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P499 = PInt<U499>; pub type N499 = NInt<U499>;
pub type U500 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P500 = PInt<U500>; pub type N500 = NInt<U500>;
pub type U501 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P501 = PInt<U501>; pub type N501 = NInt<U501>;
pub type U502 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P502 = PInt<U502>; pub type N502 = NInt<U502>;
pub type U503 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P503 = PInt<U503>; pub type N503 = NInt<U503>;
pub type U504 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P504 = PInt<U504>; pub type N504 = NInt<U504>;
pub type U505 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P505 = PInt<U505>; pub type N505 = NInt<U505>;
pub type U506 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P506 = PInt<U506>; pub type N506 = NInt<U506>;
pub type U507 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P507 = PInt<U507>; pub type N507 = NInt<U507>;
pub type U508 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P508 = PInt<U508>; pub type N508 = NInt<U508>;
pub type U509 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P509 = PInt<U509>; pub type N509 = NInt<U509>;
pub type U510 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P510 = PInt<U510>; pub type N510 = NInt<U510>;
pub type U511 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P511 = PInt<U511>; pub type N511 = NInt<U511>;
pub type U512 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P512 = PInt<U512>; pub type N512 = NInt<U512>;
pub type U513 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P513 = PInt<U513>; pub type N513 = NInt<U513>;
pub type U514 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P514 = PInt<U514>; pub type N514 = NInt<U514>;
pub type U515 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P515 = PInt<U515>; pub type N515 = NInt<U515>;
pub type U516 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P516 = PInt<U516>; pub type N516 = NInt<U516>;
pub type U517 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P517 = PInt<U517>; pub type N517 = NInt<U517>;
pub type U518 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P518 = PInt<U518>; pub type N518 = NInt<U518>;
pub type U519 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P519 = PInt<U519>; pub type N519 = NInt<U519>;
pub type U520 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P520 = PInt<U520>; pub type N520 = NInt<U520>;
pub type U521 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P521 = PInt<U521>; pub type N521 = NInt<U521>;
pub type U522 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P522 = PInt<U522>; pub type N522 = NInt<U522>;
pub type U523 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P523 = PInt<U523>; pub type N523 = NInt<U523>;
pub type U524 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P524 = PInt<U524>; pub type N524 = NInt<U524>;
pub type U525 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P525 = PInt<U525>; pub type N525 = NInt<U525>;
pub type U526 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P526 = PInt<U526>; pub type N526 = NInt<U526>;
pub type U527 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P527 = PInt<U527>; pub type N527 = NInt<U527>;
pub type U528 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P528 = PInt<U528>; pub type N528 = NInt<U528>;
pub type U529 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P529 = PInt<U529>; pub type N529 = NInt<U529>;
pub type U530 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P530 = PInt<U530>; pub type N530 = NInt<U530>;
pub type U531 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P531 = PInt<U531>; pub type N531 = NInt<U531>;
pub type U532 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P532 = PInt<U532>; pub type N532 = NInt<U532>;
pub type U533 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P533 = PInt<U533>; pub type N533 = NInt<U533>;
pub type U534 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P534 = PInt<U534>; pub type N534 = NInt<U534>;
pub type U535 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P535 = PInt<U535>; pub type N535 = NInt<U535>;
pub type U536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P536 = PInt<U536>; pub type N536 = NInt<U536>;
pub type U537 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P537 = PInt<U537>; pub type N537 = NInt<U537>;
pub type U538 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P538 = PInt<U538>; pub type N538 = NInt<U538>;
pub type U539 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P539 = PInt<U539>; pub type N539 = NInt<U539>;
pub type U540 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P540 = PInt<U540>; pub type N540 = NInt<U540>;
pub type U541 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P541 = PInt<U541>; pub type N541 = NInt<U541>;
pub type U542 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P542 = PInt<U542>; pub type N542 = NInt<U542>;
pub type U543 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P543 = PInt<U543>; pub type N543 = NInt<U543>;
pub type U544 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P544 = PInt<U544>; pub type N544 = NInt<U544>;
pub type U545 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P545 = PInt<U545>; pub type N545 = NInt<U545>;
pub type U546 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P546 = PInt<U546>; pub type N546 = NInt<U546>;
pub type U547 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P547 = PInt<U547>; pub type N547 = NInt<U547>;
pub type U548 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P548 = PInt<U548>; pub type N548 = NInt<U548>;
pub type U549 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P549 = PInt<U549>; pub type N549 = NInt<U549>;
pub type U550 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P550 = PInt<U550>; pub type N550 = NInt<U550>;
pub type U551 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P551 = PInt<U551>; pub type N551 = NInt<U551>;
pub type U552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P552 = PInt<U552>; pub type N552 = NInt<U552>;
pub type U553 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P553 = PInt<U553>; pub type N553 = NInt<U553>;
pub type U554 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P554 = PInt<U554>; pub type N554 = NInt<U554>;
pub type U555 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P555 = PInt<U555>; pub type N555 = NInt<U555>;
pub type U556 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P556 = PInt<U556>; pub type N556 = NInt<U556>;
pub type U557 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P557 = PInt<U557>; pub type N557 = NInt<U557>;
pub type U558 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P558 = PInt<U558>; pub type N558 = NInt<U558>;
pub type U559 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P559 = PInt<U559>; pub type N559 = NInt<U559>;
pub type U560 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P560 = PInt<U560>; pub type N560 = NInt<U560>;
pub type U561 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P561 = PInt<U561>; pub type N561 = NInt<U561>;
pub type U562 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P562 = PInt<U562>; pub type N562 = NInt<U562>;
pub type U563 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P563 = PInt<U563>; pub type N563 = NInt<U563>;
pub type U564 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P564 = PInt<U564>; pub type N564 = NInt<U564>;
pub type U565 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P565 = PInt<U565>; pub type N565 = NInt<U565>;
pub type U566 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P566 = PInt<U566>; pub type N566 = NInt<U566>;
pub type U567 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P567 = PInt<U567>; pub type N567 = NInt<U567>;
pub type U568 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P568 = PInt<U568>; pub type N568 = NInt<U568>;
pub type U569 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P569 = PInt<U569>; pub type N569 = NInt<U569>;
pub type U570 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P570 = PInt<U570>; pub type N570 = NInt<U570>;
pub type U571 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P571 = PInt<U571>; pub type N571 = NInt<U571>;
pub type U572 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P572 = PInt<U572>; pub type N572 = NInt<U572>;
pub type U573 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P573 = PInt<U573>; pub type N573 = NInt<U573>;
pub type U574 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P574 = PInt<U574>; pub type N574 = NInt<U574>;
pub type U575 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P575 = PInt<U575>; pub type N575 = NInt<U575>;
pub type U576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P576 = PInt<U576>; pub type N576 = NInt<U576>;
pub type U577 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P577 = PInt<U577>; pub type N577 = NInt<U577>;
pub type U578 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P578 = PInt<U578>; pub type N578 = NInt<U578>;
pub type U579 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P579 = PInt<U579>; pub type N579 = NInt<U579>;
pub type U580 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P580 = PInt<U580>; pub type N580 = NInt<U580>;
pub type U581 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P581 = PInt<U581>; pub type N581 = NInt<U581>;
pub type U582 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P582 = PInt<U582>; pub type N582 = NInt<U582>;
pub type U583 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P583 = PInt<U583>; pub type N583 = NInt<U583>;
pub type U584 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P584 = PInt<U584>; pub type N584 = NInt<U584>;
pub type U585 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P585 = PInt<U585>; pub type N585 = NInt<U585>;
pub type U586 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P586 = PInt<U586>; pub type N586 = NInt<U586>;
pub type U587 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P587 = PInt<U587>; pub type N587 = NInt<U587>;
pub type U588 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P588 = PInt<U588>; pub type N588 = NInt<U588>;
pub type U589 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P589 = PInt<U589>; pub type N589 = NInt<U589>;
pub type U590 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P590 = PInt<U590>; pub type N590 = NInt<U590>;
pub type U591 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P591 = PInt<U591>; pub type N591 = NInt<U591>;
pub type U592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P592 = PInt<U592>; pub type N592 = NInt<U592>;
pub type U593 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P593 = PInt<U593>; pub type N593 = NInt<U593>;
pub type U594 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P594 = PInt<U594>; pub type N594 = NInt<U594>;
pub type U595 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P595 = PInt<U595>; pub type N595 = NInt<U595>;
pub type U596 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P596 = PInt<U596>; pub type N596 = NInt<U596>;
pub type U597 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P597 = PInt<U597>; pub type N597 = NInt<U597>;
pub type U598 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P598 = PInt<U598>; pub type N598 = NInt<U598>;
pub type U599 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P599 = PInt<U599>; pub type N599 = NInt<U599>;
pub type U600 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P600 = PInt<U600>; pub type N600 = NInt<U600>;
pub type U601 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P601 = PInt<U601>; pub type N601 = NInt<U601>;
pub type U602 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P602 = PInt<U602>; pub type N602 = NInt<U602>;
pub type U603 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P603 = PInt<U603>; pub type N603 = NInt<U603>;
pub type U604 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P604 = PInt<U604>; pub type N604 = NInt<U604>;
pub type U605 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P605 = PInt<U605>; pub type N605 = NInt<U605>;
pub type U606 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P606 = PInt<U606>; pub type N606 = NInt<U606>;
pub type U607 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P607 = PInt<U607>; pub type N607 = NInt<U607>;
pub type U608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P608 = PInt<U608>; pub type N608 = NInt<U608>;
pub type U609 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P609 = PInt<U609>; pub type N609 = NInt<U609>;
pub type U610 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P610 = PInt<U610>; pub type N610 = NInt<U610>;
pub type U611 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P611 = PInt<U611>; pub type N611 = NInt<U611>;
pub type U612 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P612 = PInt<U612>; pub type N612 = NInt<U612>;
pub type U613 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P613 = PInt<U613>; pub type N613 = NInt<U613>;
pub type U614 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P614 = PInt<U614>; pub type N614 = NInt<U614>;
pub type U615 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P615 = PInt<U615>; pub type N615 = NInt<U615>;
pub type U616 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P616 = PInt<U616>; pub type N616 = NInt<U616>;
pub type U617 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P617 = PInt<U617>; pub type N617 = NInt<U617>;
pub type U618 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P618 = PInt<U618>; pub type N618 = NInt<U618>;
pub type U619 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P619 = PInt<U619>; pub type N619 = NInt<U619>;
pub type U620 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P620 = PInt<U620>; pub type N620 = NInt<U620>;
pub type U621 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P621 = PInt<U621>; pub type N621 = NInt<U621>;
pub type U622 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P622 = PInt<U622>; pub type N622 = NInt<U622>;
pub type U623 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P623 = PInt<U623>; pub type N623 = NInt<U623>;
pub type U624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P624 = PInt<U624>; pub type N624 = NInt<U624>;
pub type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P625 = PInt<U625>; pub type N625 = NInt<U625>;
pub type U626 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P626 = PInt<U626>; pub type N626 = NInt<U626>;
pub type U627 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P627 = PInt<U627>; pub type N627 = NInt<U627>;
pub type U628 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P628 = PInt<U628>; pub type N628 = NInt<U628>;
pub type U629 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P629 = PInt<U629>; pub type N629 = NInt<U629>;
pub type U630 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P630 = PInt<U630>; pub type N630 = NInt<U630>;
pub type U631 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P631 = PInt<U631>; pub type N631 = NInt<U631>;
pub type U632 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P632 = PInt<U632>; pub type N632 = NInt<U632>;
pub type U633 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P633 = PInt<U633>; pub type N633 = NInt<U633>;
pub type U634 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P634 = PInt<U634>; pub type N634 = NInt<U634>;
pub type U635 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P635 = PInt<U635>; pub type N635 = NInt<U635>;
pub type U636 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P636 = PInt<U636>; pub type N636 = NInt<U636>;
pub type U637 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P637 = PInt<U637>; pub type N637 = NInt<U637>;
pub type U638 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P638 = PInt<U638>; pub type N638 = NInt<U638>;
pub type U639 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P639 = PInt<U639>; pub type N639 = NInt<U639>;
pub type U640 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P640 = PInt<U640>; pub type N640 = NInt<U640>;
pub type U641 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P641 = PInt<U641>; pub type N641 = NInt<U641>;
pub type U642 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P642 = PInt<U642>; pub type N642 = NInt<U642>;
pub type U643 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P643 = PInt<U643>; pub type N643 = NInt<U643>;
pub type U644 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P644 = PInt<U644>; pub type N644 = NInt<U644>;
pub type U645 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P645 = PInt<U645>; pub type N645 = NInt<U645>;
pub type U646 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P646 = PInt<U646>; pub type N646 = NInt<U646>;
pub type U647 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P647 = PInt<U647>; pub type N647 = NInt<U647>;
pub type U648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P648 = PInt<U648>; pub type N648 = NInt<U648>;
pub type U649 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P649 = PInt<U649>; pub type N649 = NInt<U649>;
pub type U650 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P650 = PInt<U650>; pub type N650 = NInt<U650>;
pub type U651 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P651 = PInt<U651>; pub type N651 = NInt<U651>;
pub type U652 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P652 = PInt<U652>; pub type N652 = NInt<U652>;
pub type U653 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P653 = PInt<U653>; pub type N653 = NInt<U653>;
pub type U654 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P654 = PInt<U654>; pub type N654 = NInt<U654>;
pub type U655 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P655 = PInt<U655>; pub type N655 = NInt<U655>;
pub type U656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P656 = PInt<U656>; pub type N656 = NInt<U656>;
pub type U657 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P657 = PInt<U657>; pub type N657 = NInt<U657>;
pub type U658 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P658 = PInt<U658>; pub type N658 = NInt<U658>;
pub type U659 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P659 = PInt<U659>; pub type N659 = NInt<U659>;
pub type U660 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P660 = PInt<U660>; pub type N660 = NInt<U660>;
pub type U661 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P661 = PInt<U661>; pub type N661 = NInt<U661>;
pub type U662 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P662 = PInt<U662>; pub type N662 = NInt<U662>;
pub type U663 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P663 = PInt<U663>; pub type N663 = NInt<U663>;
pub type U664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P664 = PInt<U664>; pub type N664 = NInt<U664>;
pub type U665 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P665 = PInt<U665>; pub type N665 = NInt<U665>;
pub type U666 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P666 = PInt<U666>; pub type N666 = NInt<U666>;
pub type U667 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P667 = PInt<U667>; pub type N667 = NInt<U667>;
pub type U668 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P668 = PInt<U668>; pub type N668 = NInt<U668>;
pub type U669 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P669 = PInt<U669>; pub type N669 = NInt<U669>;
pub type U670 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P670 = PInt<U670>; pub type N670 = NInt<U670>;
pub type U671 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P671 = PInt<U671>; pub type N671 = NInt<U671>;
pub type U672 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P672 = PInt<U672>; pub type N672 = NInt<U672>;
pub type U673 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P673 = PInt<U673>; pub type N673 = NInt<U673>;
pub type U674 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P674 = PInt<U674>; pub type N674 = NInt<U674>;
pub type U675 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P675 = PInt<U675>; pub type N675 = NInt<U675>;
pub type U676 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P676 = PInt<U676>; pub type N676 = NInt<U676>;
pub type U677 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P677 = PInt<U677>; pub type N677 = NInt<U677>;
pub type U678 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P678 = PInt<U678>; pub type N678 = NInt<U678>;
pub type U679 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P679 = PInt<U679>; pub type N679 = NInt<U679>;
pub type U680 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P680 = PInt<U680>; pub type N680 = NInt<U680>;
pub type U681 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P681 = PInt<U681>; pub type N681 = NInt<U681>;
pub type U682 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P682 = PInt<U682>; pub type N682 = NInt<U682>;
pub type U683 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P683 = PInt<U683>; pub type N683 = NInt<U683>;
pub type U684 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P684 = PInt<U684>; pub type N684 = NInt<U684>;
pub type U685 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P685 = PInt<U685>; pub type N685 = NInt<U685>;
pub type U686 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P686 = PInt<U686>; pub type N686 = NInt<U686>;
pub type U687 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P687 = PInt<U687>; pub type N687 = NInt<U687>;
pub type U688 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P688 = PInt<U688>; pub type N688 = NInt<U688>;
pub type U689 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P689 = PInt<U689>; pub type N689 = NInt<U689>;
pub type U690 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P690 = PInt<U690>; pub type N690 = NInt<U690>;
pub type U691 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P691 = PInt<U691>; pub type N691 = NInt<U691>;
pub type U692 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P692 = PInt<U692>; pub type N692 = NInt<U692>;
pub type U693 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P693 = PInt<U693>; pub type N693 = NInt<U693>;
pub type U694 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P694 = PInt<U694>; pub type N694 = NInt<U694>;
pub type U695 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P695 = PInt<U695>; pub type N695 = NInt<U695>;
pub type U696 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P696 = PInt<U696>; pub type N696 = NInt<U696>;
pub type U697 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P697 = PInt<U697>; pub type N697 = NInt<U697>;
pub type U698 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P698 = PInt<U698>; pub type N698 = NInt<U698>;
pub type U699 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P699 = PInt<U699>; pub type N699 = NInt<U699>;
pub type U700 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P700 = PInt<U700>; pub type N700 = NInt<U700>;
pub type U701 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P701 = PInt<U701>; pub type N701 = NInt<U701>;
pub type U702 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P702 = PInt<U702>; pub type N702 = NInt<U702>;
pub type U703 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P703 = PInt<U703>; pub type N703 = NInt<U703>;
pub type U704 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P704 = PInt<U704>; pub type N704 = NInt<U704>;
pub type U705 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P705 = PInt<U705>; pub type N705 = NInt<U705>;
pub type U706 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P706 = PInt<U706>; pub type N706 = NInt<U706>;
pub type U707 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P707 = PInt<U707>; pub type N707 = NInt<U707>;
pub type U708 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P708 = PInt<U708>; pub type N708 = NInt<U708>;
pub type U709 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P709 = PInt<U709>; pub type N709 = NInt<U709>;
pub type U710 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P710 = PInt<U710>; pub type N710 = NInt<U710>;
pub type U711 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P711 = PInt<U711>; pub type N711 = NInt<U711>;
pub type U712 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P712 = PInt<U712>; pub type N712 = NInt<U712>;
pub type U713 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P713 = PInt<U713>; pub type N713 = NInt<U713>;
pub type U714 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P714 = PInt<U714>; pub type N714 = NInt<U714>;
pub type U715 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P715 = PInt<U715>; pub type N715 = NInt<U715>;
pub type U716 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P716 = PInt<U716>; pub type N716 = NInt<U716>;
pub type U717 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P717 = PInt<U717>; pub type N717 = NInt<U717>;
pub type U718 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P718 = PInt<U718>; pub type N718 = NInt<U718>;
pub type U719 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P719 = PInt<U719>; pub type N719 = NInt<U719>;
pub type U720 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P720 = PInt<U720>; pub type N720 = NInt<U720>;
pub type U721 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P721 = PInt<U721>; pub type N721 = NInt<U721>;
pub type U722 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P722 = PInt<U722>; pub type N722 = NInt<U722>;
pub type U723 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P723 = PInt<U723>; pub type N723 = NInt<U723>;
pub type U724 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P724 = PInt<U724>; pub type N724 = NInt<U724>;
pub type U725 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P725 = PInt<U725>; pub type N725 = NInt<U725>;
pub type U726 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P726 = PInt<U726>; pub type N726 = NInt<U726>;
pub type U727 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P727 = PInt<U727>; pub type N727 = NInt<U727>;
pub type U728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P728 = PInt<U728>; pub type N728 = NInt<U728>;
pub type U729 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P729 = PInt<U729>; pub type N729 = NInt<U729>;
pub type U730 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P730 = PInt<U730>; pub type N730 = NInt<U730>;
pub type U731 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P731 = PInt<U731>; pub type N731 = NInt<U731>;
pub type U732 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P732 = PInt<U732>; pub type N732 = NInt<U732>;
pub type U733 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P733 = PInt<U733>; pub type N733 = NInt<U733>;
pub type U734 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P734 = PInt<U734>; pub type N734 = NInt<U734>;
pub type U735 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P735 = PInt<U735>; pub type N735 = NInt<U735>;
pub type U736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P736 = PInt<U736>; pub type N736 = NInt<U736>;
pub type U737 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P737 = PInt<U737>; pub type N737 = NInt<U737>;
pub type U738 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P738 = PInt<U738>; pub type N738 = NInt<U738>;
pub type U739 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P739 = PInt<U739>; pub type N739 = NInt<U739>;
pub type U740 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P740 = PInt<U740>; pub type N740 = NInt<U740>;
pub type U741 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P741 = PInt<U741>; pub type N741 = NInt<U741>;
pub type U742 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P742 = PInt<U742>; pub type N742 = NInt<U742>;
pub type U743 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P743 = PInt<U743>; pub type N743 = NInt<U743>;
pub type U744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P744 = PInt<U744>; pub type N744 = NInt<U744>;
pub type U745 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P745 = PInt<U745>; pub type N745 = NInt<U745>;
pub type U746 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P746 = PInt<U746>; pub type N746 = NInt<U746>;
pub type U747 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P747 = PInt<U747>; pub type N747 = NInt<U747>;
pub type U748 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P748 = PInt<U748>; pub type N748 = NInt<U748>;
pub type U749 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P749 = PInt<U749>; pub type N749 = NInt<U749>;
pub type U750 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P750 = PInt<U750>; pub type N750 = NInt<U750>;
pub type U751 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P751 = PInt<U751>; pub type N751 = NInt<U751>;
pub type U752 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P752 = PInt<U752>; pub type N752 = NInt<U752>;
pub type U753 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P753 = PInt<U753>; pub type N753 = NInt<U753>;
pub type U754 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P754 = PInt<U754>; pub type N754 = NInt<U754>;
pub type U755 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P755 = PInt<U755>; pub type N755 = NInt<U755>;
pub type U756 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P756 = PInt<U756>; pub type N756 = NInt<U756>;
pub type U757 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P757 = PInt<U757>; pub type N757 = NInt<U757>;
pub type U758 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P758 = PInt<U758>; pub type N758 = NInt<U758>;
pub type U759 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P759 = PInt<U759>; pub type N759 = NInt<U759>;
pub type U760 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P760 = PInt<U760>; pub type N760 = NInt<U760>;
pub type U761 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P761 = PInt<U761>; pub type N761 = NInt<U761>;
pub type U762 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P762 = PInt<U762>; pub type N762 = NInt<U762>;
pub type U763 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P763 = PInt<U763>; pub type N763 = NInt<U763>;
pub type U764 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P764 = PInt<U764>; pub type N764 = NInt<U764>;
pub type U765 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P765 = PInt<U765>; pub type N765 = NInt<U765>;
pub type U766 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P766 = PInt<U766>; pub type N766 = NInt<U766>;
pub type U767 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P767 = PInt<U767>; pub type N767 = NInt<U767>;
pub type U768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P768 = PInt<U768>; pub type N768 = NInt<U768>;
pub type U769 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P769 = PInt<U769>; pub type N769 = NInt<U769>;
pub type U770 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P770 = PInt<U770>; pub type N770 = NInt<U770>;
pub type U771 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P771 = PInt<U771>; pub type N771 = NInt<U771>;
pub type U772 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P772 = PInt<U772>; pub type N772 = NInt<U772>;
pub type U773 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P773 = PInt<U773>; pub type N773 = NInt<U773>;
pub type U774 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P774 = PInt<U774>; pub type N774 = NInt<U774>;
pub type U775 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P775 = PInt<U775>; pub type N775 = NInt<U775>;
pub type U776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P776 = PInt<U776>; pub type N776 = NInt<U776>;
pub type U777 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P777 = PInt<U777>; pub type N777 = NInt<U777>;
pub type U778 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P778 = PInt<U778>; pub type N778 = NInt<U778>;
pub type U779 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P779 = PInt<U779>; pub type N779 = NInt<U779>;
pub type U780 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P780 = PInt<U780>; pub type N780 = NInt<U780>;
pub type U781 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P781 = PInt<U781>; pub type N781 = NInt<U781>;
pub type U782 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P782 = PInt<U782>; pub type N782 = NInt<U782>;
pub type U783 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P783 = PInt<U783>; pub type N783 = NInt<U783>;
pub type U784 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P784 = PInt<U784>; pub type N784 = NInt<U784>;
pub type U785 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P785 = PInt<U785>; pub type N785 = NInt<U785>;
pub type U786 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P786 = PInt<U786>; pub type N786 = NInt<U786>;
pub type U787 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P787 = PInt<U787>; pub type N787 = NInt<U787>;
pub type U788 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P788 = PInt<U788>; pub type N788 = NInt<U788>;
pub type U789 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P789 = PInt<U789>; pub type N789 = NInt<U789>;
pub type U790 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P790 = PInt<U790>; pub type N790 = NInt<U790>;
pub type U791 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P791 = PInt<U791>; pub type N791 = NInt<U791>;
pub type U792 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P792 = PInt<U792>; pub type N792 = NInt<U792>;
pub type U793 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P793 = PInt<U793>; pub type N793 = NInt<U793>;
pub type U794 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P794 = PInt<U794>; pub type N794 = NInt<U794>;
pub type U795 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P795 = PInt<U795>; pub type N795 = NInt<U795>;
pub type U796 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P796 = PInt<U796>; pub type N796 = NInt<U796>;
pub type U797 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P797 = PInt<U797>; pub type N797 = NInt<U797>;
pub type U798 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P798 = PInt<U798>; pub type N798 = NInt<U798>;
pub type U799 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P799 = PInt<U799>; pub type N799 = NInt<U799>;
pub type U800 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P800 = PInt<U800>; pub type N800 = NInt<U800>;
pub type U801 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P801 = PInt<U801>; pub type N801 = NInt<U801>;
pub type U802 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P802 = PInt<U802>; pub type N802 = NInt<U802>;
pub type U803 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P803 = PInt<U803>; pub type N803 = NInt<U803>;
pub type U804 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P804 = PInt<U804>; pub type N804 = NInt<U804>;
pub type U805 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P805 = PInt<U805>; pub type N805 = NInt<U805>;
pub type U806 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P806 = PInt<U806>; pub type N806 = NInt<U806>;
pub type U807 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P807 = PInt<U807>; pub type N807 = NInt<U807>;
pub type U808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P808 = PInt<U808>; pub type N808 = NInt<U808>;
pub type U809 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P809 = PInt<U809>; pub type N809 = NInt<U809>;
pub type U810 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P810 = PInt<U810>; pub type N810 = NInt<U810>;
pub type U811 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P811 = PInt<U811>; pub type N811 = NInt<U811>;
pub type U812 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P812 = PInt<U812>; pub type N812 = NInt<U812>;
pub type U813 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P813 = PInt<U813>; pub type N813 = NInt<U813>;
pub type U814 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P814 = PInt<U814>; pub type N814 = NInt<U814>;
pub type U815 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P815 = PInt<U815>; pub type N815 = NInt<U815>;
pub type U816 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P816 = PInt<U816>; pub type N816 = NInt<U816>;
pub type U817 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P817 = PInt<U817>; pub type N817 = NInt<U817>;
pub type U818 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P818 = PInt<U818>; pub type N818 = NInt<U818>;
pub type U819 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P819 = PInt<U819>; pub type N819 = NInt<U819>;
pub type U820 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P820 = PInt<U820>; pub type N820 = NInt<U820>;
pub type U821 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P821 = PInt<U821>; pub type N821 = NInt<U821>;
pub type U822 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P822 = PInt<U822>; pub type N822 = NInt<U822>;
pub type U823 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P823 = PInt<U823>; pub type N823 = NInt<U823>;
pub type U824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P824 = PInt<U824>; pub type N824 = NInt<U824>;
pub type U825 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P825 = PInt<U825>; pub type N825 = NInt<U825>;
pub type U826 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P826 = PInt<U826>; pub type N826 = NInt<U826>;
pub type U827 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P827 = PInt<U827>; pub type N827 = NInt<U827>;
pub type U828 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P828 = PInt<U828>; pub type N828 = NInt<U828>;
pub type U829 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P829 = PInt<U829>; pub type N829 = NInt<U829>;
pub type U830 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P830 = PInt<U830>; pub type N830 = NInt<U830>;
pub type U831 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P831 = PInt<U831>; pub type N831 = NInt<U831>;
pub type U832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P832 = PInt<U832>; pub type N832 = NInt<U832>;
pub type U833 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P833 = PInt<U833>; pub type N833 = NInt<U833>;
pub type U834 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P834 = PInt<U834>; pub type N834 = NInt<U834>;
pub type U835 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P835 = PInt<U835>; pub type N835 = NInt<U835>;
pub type U836 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P836 = PInt<U836>; pub type N836 = NInt<U836>;
pub type U837 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P837 = PInt<U837>; pub type N837 = NInt<U837>;
pub type U838 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P838 = PInt<U838>; pub type N838 = NInt<U838>;
pub type U839 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P839 = PInt<U839>; pub type N839 = NInt<U839>;
pub type U840 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P840 = PInt<U840>; pub type N840 = NInt<U840>;
pub type U841 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P841 = PInt<U841>; pub type N841 = NInt<U841>;
pub type U842 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P842 = PInt<U842>; pub type N842 = NInt<U842>;
pub type U843 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P843 = PInt<U843>; pub type N843 = NInt<U843>;
pub type U844 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P844 = PInt<U844>; pub type N844 = NInt<U844>;
pub type U845 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P845 = PInt<U845>; pub type N845 = NInt<U845>;
pub type U846 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P846 = PInt<U846>; pub type N846 = NInt<U846>;
pub type U847 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P847 = PInt<U847>; pub type N847 = NInt<U847>;
pub type U848 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P848 = PInt<U848>; pub type N848 = NInt<U848>;
pub type U849 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P849 = PInt<U849>; pub type N849 = NInt<U849>;
pub type U850 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P850 = PInt<U850>; pub type N850 = NInt<U850>;
pub type U851 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P851 = PInt<U851>; pub type N851 = NInt<U851>;
pub type U852 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P852 = PInt<U852>; pub type N852 = NInt<U852>;
pub type U853 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P853 = PInt<U853>; pub type N853 = NInt<U853>;
pub type U854 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P854 = PInt<U854>; pub type N854 = NInt<U854>;
pub type U855 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P855 = PInt<U855>; pub type N855 = NInt<U855>;
pub type U856 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P856 = PInt<U856>; pub type N856 = NInt<U856>;
pub type U857 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P857 = PInt<U857>; pub type N857 = NInt<U857>;
pub type U858 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P858 = PInt<U858>; pub type N858 = NInt<U858>;
pub type U859 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P859 = PInt<U859>; pub type N859 = NInt<U859>;
pub type U860 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P860 = PInt<U860>; pub type N860 = NInt<U860>;
pub type U861 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P861 = PInt<U861>; pub type N861 = NInt<U861>;
pub type U862 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P862 = PInt<U862>; pub type N862 = NInt<U862>;
pub type U863 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P863 = PInt<U863>; pub type N863 = NInt<U863>;
pub type U864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P864 = PInt<U864>; pub type N864 = NInt<U864>;
pub type U865 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P865 = PInt<U865>; pub type N865 = NInt<U865>;
pub type U866 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P866 = PInt<U866>; pub type N866 = NInt<U866>;
pub type U867 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P867 = PInt<U867>; pub type N867 = NInt<U867>;
pub type U868 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P868 = PInt<U868>; pub type N868 = NInt<U868>;
pub type U869 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P869 = PInt<U869>; pub type N869 = NInt<U869>;
pub type U870 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P870 = PInt<U870>; pub type N870 = NInt<U870>;
pub type U871 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P871 = PInt<U871>; pub type N871 = NInt<U871>;
pub type U872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P872 = PInt<U872>; pub type N872 = NInt<U872>;
pub type U873 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P873 = PInt<U873>; pub type N873 = NInt<U873>;
pub type U874 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P874 = PInt<U874>; pub type N874 = NInt<U874>;
pub type U875 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P875 = PInt<U875>; pub type N875 = NInt<U875>;
pub type U876 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P876 = PInt<U876>; pub type N876 = NInt<U876>;
pub type U877 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P877 = PInt<U877>; pub type N877 = NInt<U877>;
pub type U878 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P878 = PInt<U878>; pub type N878 = NInt<U878>;
pub type U879 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P879 = PInt<U879>; pub type N879 = NInt<U879>;
pub type U880 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P880 = PInt<U880>; pub type N880 = NInt<U880>;
pub type U881 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P881 = PInt<U881>; pub type N881 = NInt<U881>;
pub type U882 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P882 = PInt<U882>; pub type N882 = NInt<U882>;
pub type U883 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P883 = PInt<U883>; pub type N883 = NInt<U883>;
pub type U884 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P884 = PInt<U884>; pub type N884 = NInt<U884>;
pub type U885 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P885 = PInt<U885>; pub type N885 = NInt<U885>;
pub type U886 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P886 = PInt<U886>; pub type N886 = NInt<U886>;
pub type U887 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P887 = PInt<U887>; pub type N887 = NInt<U887>;
pub type U888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P888 = PInt<U888>; pub type N888 = NInt<U888>;
pub type U889 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P889 = PInt<U889>; pub type N889 = NInt<U889>;
pub type U890 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P890 = PInt<U890>; pub type N890 = NInt<U890>;
pub type U891 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P891 = PInt<U891>; pub type N891 = NInt<U891>;
pub type U892 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P892 = PInt<U892>; pub type N892 = NInt<U892>;
pub type U893 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P893 = PInt<U893>; pub type N893 = NInt<U893>;
pub type U894 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P894 = PInt<U894>; pub type N894 = NInt<U894>;
pub type U895 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P895 = PInt<U895>; pub type N895 = NInt<U895>;
pub type U896 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P896 = PInt<U896>; pub type N896 = NInt<U896>;
pub type U897 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P897 = PInt<U897>; pub type N897 = NInt<U897>;
pub type U898 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P898 = PInt<U898>; pub type N898 = NInt<U898>;
pub type U899 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P899 = PInt<U899>; pub type N899 = NInt<U899>;
pub type U900 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P900 = PInt<U900>; pub type N900 = NInt<U900>;
pub type U901 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P901 = PInt<U901>; pub type N901 = NInt<U901>;
pub type U902 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P902 = PInt<U902>; pub type N902 = NInt<U902>;
pub type U903 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P903 = PInt<U903>; pub type N903 = NInt<U903>;
pub type U904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P904 = PInt<U904>; pub type N904 = NInt<U904>;
pub type U905 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P905 = PInt<U905>; pub type N905 = NInt<U905>;
pub type U906 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P906 = PInt<U906>; pub type N906 = NInt<U906>;
pub type U907 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P907 = PInt<U907>; pub type N907 = NInt<U907>;
pub type U908 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P908 = PInt<U908>; pub type N908 = NInt<U908>;
pub type U909 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P909 = PInt<U909>; pub type N909 = NInt<U909>;
pub type U910 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P910 = PInt<U910>; pub type N910 = NInt<U910>;
pub type U911 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P911 = PInt<U911>; pub type N911 = NInt<U911>;
pub type U912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P912 = PInt<U912>; pub type N912 = NInt<U912>;
pub type U913 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P913 = PInt<U913>; pub type N913 = NInt<U913>;
pub type U914 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P914 = PInt<U914>; pub type N914 = NInt<U914>;
pub type U915 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P915 = PInt<U915>; pub type N915 = NInt<U915>;
pub type U916 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P916 = PInt<U916>; pub type N916 = NInt<U916>;
pub type U917 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P917 = PInt<U917>; pub type N917 = NInt<U917>;
pub type U918 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P918 = PInt<U918>; pub type N918 = NInt<U918>;
pub type U919 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P919 = PInt<U919>; pub type N919 = NInt<U919>;
pub type U920 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P920 = PInt<U920>; pub type N920 = NInt<U920>;
pub type U921 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P921 = PInt<U921>; pub type N921 = NInt<U921>;
pub type U922 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P922 = PInt<U922>; pub type N922 = NInt<U922>;
pub type U923 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P923 = PInt<U923>; pub type N923 = NInt<U923>;
pub type U924 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P924 = PInt<U924>; pub type N924 = NInt<U924>;
pub type U925 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P925 = PInt<U925>; pub type N925 = NInt<U925>;
pub type U926 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P926 = PInt<U926>; pub type N926 = NInt<U926>;
pub type U927 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P927 = PInt<U927>; pub type N927 = NInt<U927>;
pub type U928 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P928 = PInt<U928>; pub type N928 = NInt<U928>;
pub type U929 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P929 = PInt<U929>; pub type N929 = NInt<U929>;
pub type U930 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P930 = PInt<U930>; pub type N930 = NInt<U930>;
pub type U931 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P931 = PInt<U931>; pub type N931 = NInt<U931>;
pub type U932 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P932 = PInt<U932>; pub type N932 = NInt<U932>;
pub type U933 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P933 = PInt<U933>; pub type N933 = NInt<U933>;
pub type U934 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P934 = PInt<U934>; pub type N934 = NInt<U934>;
pub type U935 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P935 = PInt<U935>; pub type N935 = NInt<U935>;
pub type U936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P936 = PInt<U936>; pub type N936 = NInt<U936>;
pub type U937 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P937 = PInt<U937>; pub type N937 = NInt<U937>;
pub type U938 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P938 = PInt<U938>; pub type N938 = NInt<U938>;
pub type U939 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P939 = PInt<U939>; pub type N939 = NInt<U939>;
pub type U940 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P940 = PInt<U940>; pub type N940 = NInt<U940>;
pub type U941 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P941 = PInt<U941>; pub type N941 = NInt<U941>;
pub type U942 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P942 = PInt<U942>; pub type N942 = NInt<U942>;
pub type U943 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P943 = PInt<U943>; pub type N943 = NInt<U943>;
pub type U944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P944 = PInt<U944>; pub type N944 = NInt<U944>;
pub type U945 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P945 = PInt<U945>; pub type N945 = NInt<U945>;
pub type U946 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P946 = PInt<U946>; pub type N946 = NInt<U946>;
pub type U947 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P947 = PInt<U947>; pub type N947 = NInt<U947>;
pub type U948 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P948 = PInt<U948>; pub type N948 = NInt<U948>;
pub type U949 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P949 = PInt<U949>; pub type N949 = NInt<U949>;
pub type U950 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P950 = PInt<U950>; pub type N950 = NInt<U950>;
pub type U951 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P951 = PInt<U951>; pub type N951 = NInt<U951>;
pub type U952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P952 = PInt<U952>; pub type N952 = NInt<U952>;
pub type U953 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P953 = PInt<U953>; pub type N953 = NInt<U953>;
pub type U954 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P954 = PInt<U954>; pub type N954 = NInt<U954>;
pub type U955 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P955 = PInt<U955>; pub type N955 = NInt<U955>;
pub type U956 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P956 = PInt<U956>; pub type N956 = NInt<U956>;
pub type U957 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P957 = PInt<U957>; pub type N957 = NInt<U957>;
pub type U958 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P958 = PInt<U958>; pub type N958 = NInt<U958>;
pub type U959 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P959 = PInt<U959>; pub type N959 = NInt<U959>;
pub type U960 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P960 = PInt<U960>; pub type N960 = NInt<U960>;
pub type U961 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P961 = PInt<U961>; pub type N961 = NInt<U961>;
pub type U962 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P962 = PInt<U962>; pub type N962 = NInt<U962>;
pub type U963 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P963 = PInt<U963>; pub type N963 = NInt<U963>;
pub type U964 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P964 = PInt<U964>; pub type N964 = NInt<U964>;
pub type U965 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P965 = PInt<U965>; pub type N965 = NInt<U965>;
pub type U966 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P966 = PInt<U966>; pub type N966 = NInt<U966>;
pub type U967 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P967 = PInt<U967>; pub type N967 = NInt<U967>;
pub type U968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P968 = PInt<U968>; pub type N968 = NInt<U968>;
pub type U969 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P969 = PInt<U969>; pub type N969 = NInt<U969>;
pub type U970 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P970 = PInt<U970>; pub type N970 = NInt<U970>;
pub type U971 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P971 = PInt<U971>; pub type N971 = NInt<U971>;
pub type U972 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P972 = PInt<U972>; pub type N972 = NInt<U972>;
pub type U973 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P973 = PInt<U973>; pub type N973 = NInt<U973>;
pub type U974 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P974 = PInt<U974>; pub type N974 = NInt<U974>;
pub type U975 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P975 = PInt<U975>; pub type N975 = NInt<U975>;
pub type U976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P976 = PInt<U976>; pub type N976 = NInt<U976>;
pub type U977 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P977 = PInt<U977>; pub type N977 = NInt<U977>;
pub type U978 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P978 = PInt<U978>; pub type N978 = NInt<U978>;
pub type U979 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P979 = PInt<U979>; pub type N979 = NInt<U979>;
pub type U980 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P980 = PInt<U980>; pub type N980 = NInt<U980>;
pub type U981 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P981 = PInt<U981>; pub type N981 = NInt<U981>;
pub type U982 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P982 = PInt<U982>; pub type N982 = NInt<U982>;
pub type U983 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P983 = PInt<U983>; pub type N983 = NInt<U983>;
pub type U984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P984 = PInt<U984>; pub type N984 = NInt<U984>;
pub type U985 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P985 = PInt<U985>; pub type N985 = NInt<U985>;
pub type U986 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P986 = PInt<U986>; pub type N986 = NInt<U986>;
pub type U987 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P987 = PInt<U987>; pub type N987 = NInt<U987>;
pub type U988 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P988 = PInt<U988>; pub type N988 = NInt<U988>;
pub type U989 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P989 = PInt<U989>; pub type N989 = NInt<U989>;
pub type U990 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P990 = PInt<U990>; pub type N990 = NInt<U990>;
pub type U991 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P991 = PInt<U991>; pub type N991 = NInt<U991>;
pub type U992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P992 = PInt<U992>; pub type N992 = NInt<U992>;
pub type U993 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P993 = PInt<U993>; pub type N993 = NInt<U993>;
pub type U994 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P994 = PInt<U994>; pub type N994 = NInt<U994>;
pub type U995 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P995 = PInt<U995>; pub type N995 = NInt<U995>;
pub type U996 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P996 = PInt<U996>; pub type N996 = NInt<U996>;
pub type U997 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P997 = PInt<U997>; pub type N997 = NInt<U997>;
pub type U998 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P998 = PInt<U998>; pub type N998 = NInt<U998>;
pub type U999 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P999 = PInt<U999>; pub type N999 = NInt<U999>;
pub type U1000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1000 = PInt<U1000>; pub type N1000 = NInt<U1000>;
pub type U1001 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1001 = PInt<U1001>; pub type N1001 = NInt<U1001>;
pub type U1002 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1002 = PInt<U1002>; pub type N1002 = NInt<U1002>;
pub type U1003 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1003 = PInt<U1003>; pub type N1003 = NInt<U1003>;
pub type U1004 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1004 = PInt<U1004>; pub type N1004 = NInt<U1004>;
pub type U1005 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1005 = PInt<U1005>; pub type N1005 = NInt<U1005>;
pub type U1006 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1006 = PInt<U1006>; pub type N1006 = NInt<U1006>;
pub type U1007 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1007 = PInt<U1007>; pub type N1007 = NInt<U1007>;
pub type U1008 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
```

```
pub type P1008 = PInt<U1008>; pub type N1008 = NInt<U1008>;
pub type U1009 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1009 = PInt<U1009>; pub type N1009 = NInt<U1009>;
pub type U1010 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1010 = PInt<U1010>; pub type N1010 = NInt<U1010>;
pub type U1011 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1011 = PInt<U1011>; pub type N1011 = NInt<U1011>;
pub type U1012 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1012 = PInt<U1012>; pub type N1012 = NInt<U1012>;
pub type U1013 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1013 = PInt<U1013>; pub type N1013 = NInt<U1013>;
pub type U1014 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1014 = PInt<U1014>; pub type N1014 = NInt<U1014>;
pub type U1015 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1015 = PInt<U1015>; pub type N1015 = NInt<U1015>;
pub type U1016 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1016 = PInt<U1016>; pub type N1016 = NInt<U1016>;
pub type U1017 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1017 = PInt<U1017>; pub type N1017 = NInt<U1017>;
pub type U1018 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1018 = PInt<U1018>; pub type N1018 = NInt<U1018>;
pub type U1019 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1019 = PInt<U1019>; pub type N1019 = NInt<U1019>;
pub type U1020 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1020 = PInt<U1020>; pub type N1020 = NInt<U1020>;
pub type U1021 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1021 = PInt<U1021>; pub type N1021 = NInt<U1021>;
pub type U1022 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1022 = PInt<U1022>; pub type N1022 = NInt<U1022>;
pub type U1023 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1023 = PInt<U1023>; pub type N1023 = NInt<U1023>;
pub type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1024 = PInt<U1024>; pub type N1024 = NInt<U1024>;
pub type U2048 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2048 = PInt<U2048>; pub type N2048 = NInt<U2048>;
pub type U4096 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4096 = PInt<U4096>; pub type N4096 = NInt<U4096>;
pub type U8192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P8192 = PInt<U8192>; pub type N8192 = NInt<U8192>;
pub type U16384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P16384 = PInt<U16384>; pub type N16384 = NInt<U16384>;
pub type U32768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P32768 = PInt<U32768>; pub type N32768 = NInt<U32768>;
pub type U65536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P65536 = PInt<U65536>; pub type N65536 = NInt<U65536>;
pub type U131072 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P131072 = PInt<U131072>; pub type N131072 = NInt<U131072>;
pub type U262144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P262144 = PInt<U262144>; pub type N262144 = NInt<U262144>;
pub type U524288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P524288 = PInt<U524288>; pub type N524288 = NInt<U524288>;
pub type U1048576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
```

```
pub type P1048576 = PInt<U1048576>; pub type N1048576 = NInt<U1048576>;
pub type U2097152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P2097152 = PInt<U2097152>; pub type N2097152 = NInt<U2097152>;
pub type U4194304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P4194304 = PInt<U4194304>; pub type N4194304 = NInt<U4194304>;
pub type U8388608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P8388608 = PInt<U8388608>; pub type N8388608 = NInt<U8388608>;
pub type U16777216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P16777216 = PInt<U16777216>; pub type N16777216 = NInt<U167772
pub type U33554432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P33554432 = PInt<U33554432>; pub type N33554432 = NInt<U335544
pub type U67108864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P67108864 = PInt<U67108864>; pub type N67108864 = NInt<U671088
pub type U134217728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P134217728 = PInt<U134217728>; pub type N134217728 = NInt<U134
pub type U268435456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P268435456 = PInt<U268435456>; pub type N268435456 = NInt<U268
pub type U536870912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P536870912 = PInt<U536870912>; pub type N536870912 = NInt<U536
pub type U1073741824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1073741824 = PInt<U1073741824>; pub type N1073741824 = NInt<U
pub type U2147483648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P2147483648 = PInt<U2147483648>; pub type N2147483648 = NInt<U
pub type U4294967296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P4294967296 = PInt<U4294967296>; pub type N4294967296 = NInt<U
pub type U8589934592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P8589934592 = PInt<U8589934592>; pub type N8589934592 = NInt<U
pub type U17179869184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P17179869184 = PInt<U17179869184>; pub type N17179869184 = NIn
pub type U34359738368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P34359738368 = PInt<U34359738368>; pub type N34359738368 = NIn
pub type U68719476736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P68719476736 = PInt<U68719476736>; pub type N68719476736 = NIn
pub type U137438953472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P137438953472 = PInt<U137438953472>; pub type N137438953472 =
pub type U274877906944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P274877906944 = PInt<U274877906944>; pub type N274877906944 =
pub type U549755813888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P549755813888 = PInt<U549755813888>; pub type N549755813888 =
pub type U1099511627776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1099511627776 = PInt<U1099511627776>; pub type N1099511627776
pub type U2199023255552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P2199023255552 = PInt<U2199023255552>; pub type N2199023255552
pub type U4398046511104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P4398046511104 = PInt<U4398046511104>; pub type N4398046511104
pub type U8796093022208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P8796093022208 = PInt<U8796093022208>; pub type N8796093022208
pub type U17592186044416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P17592186044416 = PInt<U17592186044416>; pub type N17592186044
pub type U35184372088832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P35184372088832 = PInt<U35184372088832>; pub type N35184372088
pub type U70368744177664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
```

```
pub type P70368744177664 = PInt<U70368744177664>; pub type N7036874417
pub type U140737488355328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P140737488355328 = PInt<U140737488355328>; pub type N140737488
pub type U281474976710656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P281474976710656 = PInt<U281474976710656>; pub type N281474976
pub type U562949953421312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P562949953421312 = PInt<U562949953421312>; pub type N562949953
pub type U1125899906842624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P1125899906842624 = PInt<U1125899906842624>; pub type N1125899
pub type U2251799813685248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P2251799813685248 = PInt<U2251799813685248>; pub type N2251799
pub type U4503599627370496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P4503599627370496 = PInt<U4503599627370496>; pub type N4503599
pub type U9007199254740992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P9007199254740992 = PInt<U9007199254740992>; pub type N9007199
pub type U18014398509481984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P18014398509481984 = PInt<U18014398509481984>; pub type N18014
pub type U36028797018963968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P36028797018963968 = PInt<U36028797018963968>; pub type N36028
pub type U72057594037927936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P72057594037927936 = PInt<U72057594037927936>; pub type N72057
pub type U144115188075855872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P144115188075855872 = PInt<U144115188075855872>; pub type N144
pub type U288230376151711744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P288230376151711744 = PInt<U288230376151711744>; pub type N288
pub type U576460752303423488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P576460752303423488 = PInt<U576460752303423488>; pub type N576
pub type U1152921504606846976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1152921504606846976 = PInt<U1152921504606846976>; pub type N1
pub type U2305843009213693952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2305843009213693952 = PInt<U2305843009213693952>; pub type N2
pub type U4611686018427387904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4611686018427387904 = PInt<U4611686018427387904>; pub type N4
pub type U9223372036854775808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type U10000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P10000 = PInt<U10000>; pub type N10000 = NInt<U10000>;
pub type U100000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P100000 = PInt<U100000>; pub type N100000 = NInt<U100000>;
pub type U1000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1000000 = PInt<U1000000>; pub type N1000000 = NInt<U1000000>;
pub type U10000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P10000000 = PInt<U10000000>; pub type N10000000 = NInt<U100000
pub type U100000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P100000000 = PInt<U100000000>; pub type N100000000 = NInt<U100
pub type U1000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1000000000 = PInt<U1000000000>; pub type N1000000000 = NInt<U
pub type U10000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P10000000000 = PInt<U10000000000>; pub type N10000000000 = NIn
pub type U100000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P100000000000 = PInt<U100000000000>; pub type N100000000000 =
pub type U1000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1000000000000 = PInt<U1000000000000>; pub type N1000000000000
```

```
    pub type U10000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P10000000000000 = PInt<U10000000000000>; pub type N10000000000
    pub type U100000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P100000000000000 = PInt<U100000000000000>; pub type N100000000
    pub type U1000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P1000000000000000 = PInt<U1000000000000000>; pub type N1000000
    pub type U10000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P10000000000000000 = PInt<U10000000000000000>; pub type N10000
    pub type U100000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P100000000000000000 = PInt<U100000000000000000>; pub type N100
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P1000000000000000000 = PInt<U1000000000000000000>; pub type N1
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
}
```

File: ./target/x86_64-pc-windows-gnu/release/build/typenum-10a9d0

```
extern crate typenum;

use std::ops::*;
use std::cmp::Ordering;
use typenum::*;

#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitOrU0 = <<A as BitOr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitOrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_BitXor_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitXorU0 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitXorU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU0 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU0 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0AddU0 = <<A as Add<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0AddU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MaxU0 = <<A as Max<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MaxU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0GcdU0 = <<A as Gcd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0GcdU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Sub_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0SubU0 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0SubU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_0() {
    type A = UTerm;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_0() {
    type A = UTerm;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U0CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_1() {
    type A = UTerm;
```

```
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU1 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitXorU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU1 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU1 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0AddU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0MinU1 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU1 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU1 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU1 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU2 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitXorU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU2 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU2 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0AddU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU2 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_Mul_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU2 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0PowU2 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU3 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitXorU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU3 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU3 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0AddU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU3 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_3() {
    type A = UTerm;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU3 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU3 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_BitOr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU4 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitXorU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU4 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U0AddU4 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0AddU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU4 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU4 = <<A as Mul<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0MulU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU4 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_Cmp_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU5 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitXorU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0ShlU5 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU5 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0AddU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU5 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU5 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_5() {
    type A = UTerm;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU5 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU0 = <<A as BitOr<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U1BitOrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU0 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitXorU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShlU0 = <<A as Shl<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShlU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShrU0 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU0 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1AddU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Min_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU0 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU0 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1SubU0 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1SubU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U1CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitXorU1 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitXorU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU1 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1ShlU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU1 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1AddU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_1() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1SubU1 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1SubU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U1MulU1 = <<A as Mul<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MulU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1DivU1 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1DivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_PartialDiv_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U1PartialDivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU1 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU2 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitXorU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U1ShlU2 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1ShlU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU2 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1AddU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU2 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U1MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU2 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MulU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Pow_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU2 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U1BitXorU3 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1BitXorU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU3 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U1ShlU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU3 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1AddU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU3 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_Max_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU3 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MulU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_3() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU3 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U1BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU4 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitXorU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU4 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U1ShlU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU4 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1AddU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Min_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU4 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU4 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MulU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU4 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU5 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1BitXorU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U1ShlU5 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U1ShlU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_5() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU5 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U1AddU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU5 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U1MulU5 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MulU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU5 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU5 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU0 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU0 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitXorU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU0 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShlU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2ShrU0 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU0 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2AddU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU0 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU0 = <<A as Gcd<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2GcdU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2SubU0 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2SubU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U2CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU1 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitXorU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU1 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2ShlU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU1 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2AddU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU1 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2SubU1 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2SubU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU1 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MulU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2DivU1 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2DivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U2PartialDivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU1 = <<A as Pow<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2PowU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitXorU2 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitXorU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU2 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2ShlU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU2 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2AddU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2SubU2 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2SubU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2MulU2 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MulU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U2PartialDivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU2 = <<A as Pow<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U2PowU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU3 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU3 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2BitXorU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU3 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2ShlU3 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU3 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2AddU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU3 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU3 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2MulU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU3 = <<A as Pow<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2PowU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU4 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitOrU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2BitXorU4 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitXorU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU4 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2ShlU4 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU4 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2AddU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU4 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2MinU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU4 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU4 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2MulU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU4 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU4 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2PowU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU5 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitXorU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU5 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U2ShlU5 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU5 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2AddU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU5 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU5 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U2MulU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU5 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2PowU5 = <<A as Pow<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2PowU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU0 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU0 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitXorU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShlU0 = <<A as Shl<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShlU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU0 = <<A as Shr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU0 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3AddU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU0 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU0 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3SubU0 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3SubU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U3CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU1 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitXorU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU1 = <<A as Shl<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3ShlU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU1 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3AddU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU1 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3SubU1 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3SubU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU1 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MulU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU1 = <<A as Div<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3DivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U3>>::Outpu

    assert_eq!(<U3PartialDivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU1 = <<A as Pow<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3PowU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3BitXorU2 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitXorU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU2 = <<A as Shl<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3ShlU2 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU2 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3AddU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U3MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU2 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU2 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3SubU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU2 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3MulU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU2 = <<A as Pow<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3PowU2 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U3BitAndU3 = <<A as BitAnd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitAndU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitXorU3 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitXorU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU3 = <<A as Shl<B>>::Output as Same<U24>>::Output;

    assert_eq!(<U3ShlU3 as Unsigned>::to_u64(), <U24 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU3 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3AddU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3SubU3 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3SubU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU3 = <<A as Mul<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3MulU3 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U3PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U3PartialDivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU3 = <<A as Pow<B>>::Output as Same<U27>>::Output;

    assert_eq!(<U3PowU3 as Unsigned>::to_u64(), <U27 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU4 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU4 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitXorU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U48 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU4 = <<A as Shl<B>>::Output as Same<U48>>::Output;

    assert_eq!(<U3ShlU4 as Unsigned>::to_u64(), <U48 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU4 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3AddU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3MinU4 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU4 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3MulU4 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU4 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U3PowU4 = <<A as Pow<B>>::Output as Same<U81>>::Output;

    assert_eq!(<U3PowU4 as Unsigned>::to_u64(), <U81 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU5 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3BitXorU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU5 = <<A as Shl<B>>::Output as Same<U96>>::Output;

    assert_eq!(<U3ShlU5 as Unsigned>::to_u64(), <U96 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3AddU5 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U3AddU5 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU5 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU5 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U3MulU5 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU5 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U3PowU5 = <<A as Pow<B>>::Output as Same<U243>>::Output;

    assert_eq!(<U3PowU5 as Unsigned>::to_u64(), <U243 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU0 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU0 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitXorU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU0 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShlU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU0 = <<A as Shr<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4ShrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU0 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4AddU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU0 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU0 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU0 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4SubU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U4CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U4BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU1 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitXorU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU1 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4ShlU1 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU1 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4AddU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU1 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU1 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4SubU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU1 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MulU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU1 = <<A as Div<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4DivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U4PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U4>>::Outpu

    assert_eq!(<U4PartialDivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU1 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4PowU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU2 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitOrU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU2 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitXorU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU2 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4ShlU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU2 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4AddU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU2 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU2 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4SubU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU2 = <<A as Mul<B>>::Output as Same<U8>>::Output;
```

```
    assert_eq!(<U4MulU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U4PartialDivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU2 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4PowU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU3 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitXorU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
```

```
    type U4ShlU3 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U4ShlU3 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU3 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4AddU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU3 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU3 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4SubU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU3 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U4MulU3 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4PowU3 = <<A as Pow<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4PowU3 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitXorU4 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitXorU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU4 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4ShlU4 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU4 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4AddU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4SubU4 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4SubU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MulU4 = <<A as Mul<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4MulU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U4PartialDivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0

    #[allow(non_camel_case_types)]
    type U4PowU4 = <<A as Pow<B>>::Output as Same<U256>>::Output;
```

```
    assert_eq!(<U4PowU4 as Unsigned>::to_u64(), <U256 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU5 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU5 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4BitXorU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0

    #[allow(non_camel_case_types)]
    type U4ShlU5 = <<A as Shl<B>>::Output as Same<U128>>::Output;

    assert_eq!(<U4ShlU5 as Unsigned>::to_u64(), <U128 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU5 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U4AddU5 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU5 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U4MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU5 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U4MulU5 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4RemU5 = <<A as Rem<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4RemU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTe

    #[allow(non_camel_case_types)]
    type U4PowU5 = <<A as Pow<B>>::Output as Same<U1024>>::Output;

    assert_eq!(<U4PowU5 as Unsigned>::to_u64(), <U1024 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU0 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitXorU0 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitXorU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShlU0 = <<A as Shl<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShlU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU0 = <<A as Shr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU0 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5AddU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U5MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU0 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU0 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU0 = <<A as Sub<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5SubU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U5CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U5BitXorU1 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitXorU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU1 = <<A as Shl<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5ShlU1 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU1 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5AddU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU1 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU1 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5SubU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU1 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MulU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU1 = <<A as Div<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5DivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U5>>::Outpu

    assert_eq!(<U5PartialDivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU1 = <<A as Pow<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5PowU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU1 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<U5CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU2 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU2 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitXorU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU2 = <<A as Shl<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5ShlU2 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU2 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5AddU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU2 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU2 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5SubU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU2 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5MulU2 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U5RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU2 = <<A as Pow<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5PowU2 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU3 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5BitXorU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU3 = <<A as Shl<B>>::Output as Same<U40>>::Output;

    assert_eq!(<U5ShlU3 as Unsigned>::to_u64(), <U40 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU3 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U5AddU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U5MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU3 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU3 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5SubU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU3 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U5MulU3 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U5PowU3 = <<A as Pow<B>>::Output as Same<U125>>::Output;

    assert_eq!(<U5PowU3 as Unsigned>::to_u64(), <U125 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU4 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitXorU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU4 = <<A as Shl<B>>::Output as Same<U80>>::Output;

    assert_eq!(<U5ShlU4 as Unsigned>::to_u64(), <U80 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U5ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU4 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U5AddU4 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU4 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU4 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5SubU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU4 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5MulU4 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1
```

```
    #[allow(non_camel_case_types)]
    type U5PowU4 = <<A as Pow<B>>::Output as Same<U625>>::Output;

    assert_eq!(<U5PowU4 as Unsigned>::to_u64(), <U625 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU5 = <<A as BitAnd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitAndU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitXorU5 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitXorU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1

    #[allow(non_camel_case_types)]
    type U5ShlU5 = <<A as Shl<B>>::Output as Same<U160>>::Output;

    assert_eq!(<U5ShlU5 as Unsigned>::to_u64(), <U160 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU5 = <<A as Add<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5AddU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU5 = <<A as Min<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MinU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5SubU5 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5SubU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U25 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU5 = <<A as Mul<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5MulU5 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U5DivU5 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U5PartialDivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type U5PowU5 = <<A as Pow<B>>::Output as Same<U3125>>::Output;

    assert_eq!(<U5PowU5 as Unsigned>::to_u64(), <U3125 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN5 = <<A as Add<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5AddN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5SubN5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5SubN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulN5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5MulN5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MaxN5 = <<A as Max<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MaxN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu
```

```
    assert_eq!(<N5PartialDivN5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN4 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5AddN4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5SubN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulN4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N5MulN4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN3 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5AddN3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5SubN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N5MulN3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N5RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN2 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5AddN2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5SubN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulN2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N5MulN2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN1 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5AddN1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5SubN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN1 = <<A as Mul<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<N5MulN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5DivN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<N5PartialDivN1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Add_0 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Add_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5Sub_0 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Sub_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Min_0 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Min_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N5Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5AddP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP1 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5SubP1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MulP1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MulP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP1 = <<A as Div<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5DivP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<N5PartialDivP1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PowP1 = <<A as Pow<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5PowP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5AddP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP2 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5SubP2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulP2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5MulP2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N5MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5AddP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP3 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5SubP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N5MulP3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MinP3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N5MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N5RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type N5PowP3 = <<A as Pow<B>>::Output as Same<N125>>::Output;

    assert_eq!(<N5PowP3 as Integer>::to_i64(), <N125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5AddP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP4 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5SubP4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulP4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N5MulP4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N5MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type N5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<N5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5AddP5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5AddP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP5 = <<A as Sub<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5SubP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulP5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<N5MulP5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N5PartialDivP5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N5PowP5 = <<A as Pow<B>>::Output as Same<N3125>>::Output;
```

```
    assert_eq!(<N5PowP5 as Integer>::to_i64(), <N3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN5 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4AddN5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN5 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4SubN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N4MulN5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN5 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N4RemN5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN4 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4AddN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4SubN4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4SubN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4MulN4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N4_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N4PartialDivN4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN3 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4AddN3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4SubN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N4MulN3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN2 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4AddN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubN2 = <<A as Sub<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N4SubN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N4MulN2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N4_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N4PartialDivN2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4AddN1 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4AddN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4SubN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MulN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4DivN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<N4PartialDivN1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Add_0 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Add_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Sub_0 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Sub_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Min_0 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4Min_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N4Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4AddP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP1 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4SubP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MulP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4DivP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<N4PartialDivP1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PowP1 = <<A as Pow<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4PowP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddP2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4AddP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP2 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4SubP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MulP2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4MulP2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N4DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N4PartialDivP2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4AddP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP3 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4SubP3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N4MulP3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N4MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N64 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP3 = <<A as Pow<B>>::Output as Same<N64>>::Output;

    assert_eq!(<N4PowP3 as Integer>::to_i64(), <N64 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4AddP4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4AddP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP4 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4SubP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<N4MulP4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MinP4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N4PartialDivP4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type N4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<N4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP5 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4AddP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP5 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4SubP5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N4MulP5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP5 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemP5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1024 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N4PowP5 = <<A as Pow<B>>::Output as Same<N1024>>::Output;

    assert_eq!(<N4PowP5 as Integer>::to_i64(), <N1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN5 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3AddN5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN5 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3SubN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N3MulN5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxN5 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN4 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3AddN4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3SubN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N3MulN4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN4 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN4 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<N3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN3 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3AddN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3SubN3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3SubN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3MulN3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N3PartialDivN3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN2 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3AddN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3SubN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N3MulN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3AddN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3SubN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN1 = <<A as Mul<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<N3MulN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3DivN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<N3PartialDivN1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Add_0 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Add_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N3Sub_0 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Sub_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Min_0 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Min_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N3Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3AddP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3SubP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MulP1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MulP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP1 = <<A as Div<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3DivP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<N3PartialDivP1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP1 = <<A as Pow<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3PowP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3AddP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP2 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3SubP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3MulP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N3MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3AddP3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3AddP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP3 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3SubP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N3MulP3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinP3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N3PartialDivP3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N27 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP3 = <<A as Pow<B>>::Output as Same<N27>>::Output;

    assert_eq!(<N3PowP3 as Integer>::to_i64(), <N27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3AddP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP4 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3SubP4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N3MulP4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP4 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<N3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP5 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3AddP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP5 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3SubP5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N3MulP5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP5 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N243 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type N3PowP5 = <<A as Pow<B>>::Output as Same<N243>>::Output;
```

```
    assert_eq!(<N3PowP5 as Integer>::to_i64(), <N243 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN5 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2AddN5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN5 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2SubN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N2MulN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN5 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N2RemN5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN4 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2AddN4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubN4 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2SubN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N2MulN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN4 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2RemN4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN3 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2AddN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2SubN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N2MulN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN3 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN2 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2AddN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2SubN2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2SubN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN2 = <<A as Mul<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<N2MulN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N2PartialDivN2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2AddN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2SubN1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2SubN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MulN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivN1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2DivN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N2PartialDivN1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2Add_0 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Add_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Sub_0 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Sub_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Min_0 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Min_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N2Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2AddP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2SubP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MulP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivP1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2DivP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N2PartialDivP1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP1 = <<A as Pow<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2PowP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2AddP2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2AddP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP2 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2SubP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MulP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MinP2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N2PartialDivP2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2AddP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP3 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2SubP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2MulP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP3 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP3 = <<A as Pow<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2PowP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddP4 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2AddP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP4 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2SubP4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2MulP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP4 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;
```

```
    assert_eq!(<N2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP5 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2AddP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP5 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2SubP5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N2MulP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP5 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N2RemP5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N32 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP5 = <<A as Pow<B>>::Output as Same<N32>>::Output;

    assert_eq!(<N2PowP5 as Integer>::to_i64(), <N32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN5 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1AddN5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN5 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1SubN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Mul_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MulN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN5 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN4 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1AddN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN4 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1SubN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MulN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN4 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N4() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN4 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<N1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN3 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1AddN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1SubN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MulN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Max_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN3 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1PowN3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1AddN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1SubN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MulN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN2 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N2() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1AddN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1SubN1 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N1SubN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MulN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Div_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivN1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1DivN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N1PartialDivN1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Add_0 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Add_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Sub_0 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Sub_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Min_0 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Min_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Max__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N1Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1AddP1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1AddP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1SubP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MulP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivP1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1DivP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N1PartialDivP1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Pow_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1AddP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1SubP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N1MulP2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MulP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP2 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1AddP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N1SubP3 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1SubP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MulP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP3 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P4() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP4 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1AddP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP4 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1SubP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MulP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP4 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP5 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1AddP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP5 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1SubP5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MulP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1MinP5 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP5 = <<A as Rem<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N1RemP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN5 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0AddN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN5 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0SubN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N5() {
    type A = Z0;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN5 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type _0DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN4 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0AddN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Sub_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN4 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0SubN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN4 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type _0GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN3 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0AddN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN3 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0SubN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N3() {
    type A = Z0;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN3 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type _0PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0AddN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0SubN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Min_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN2 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0AddN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0SubN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN1 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N1() {
    type A = Z0;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Add_0 = <<A as Add<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0Add_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Sub_0 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Sub_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Gcd__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Gcd_0 = <<A as Gcd<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Gcd_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow__0() {
    type A = Z0;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp__0() {
    type A = Z0;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type _0Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<_0Cmp_0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0AddP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0SubP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP1 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP1 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P1() {
    type A = Z0;
```

```
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0AddP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0SubP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP2 = <<A as Min<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MinP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_PartialDiv_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP2 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP3 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0AddP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP3 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0SubP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP3 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP3 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P3() {
    type A = Z0;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP4 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0AddP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP4 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0SubP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP4 = <<A as Min<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MinP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_PartialDiv_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP4 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP5 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0AddP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP5 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0SubP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP5 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP5 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P5() {
    type A = Z0;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN5 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1AddN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN5 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1SubN5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MulN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<P1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN5 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Pow_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN4 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1AddN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN4 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1SubN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P1MulN4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MulN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN4 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1AddN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P1SubN3 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1SubN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MulN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN3 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N2() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1AddN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1SubN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MulN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1MaxN2 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1AddN1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1AddN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1SubN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MulN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivN1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1DivN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P1PartialDivN1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Add_0 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Add_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub__0() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Sub_0 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Sub_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Max_0 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Max_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P1Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1AddP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1SubP1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1SubP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Mul_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MulP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1DivP1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1DivP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P1PartialDivP1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP1 as Ord>::to_ordering(), Ordering::Equal);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1AddP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1SubP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MulP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP2 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P2() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP3 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1AddP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1SubP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MulP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Min_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP3 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP4 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1AddP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP4 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1SubP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MulP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP4 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P4() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP5 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P1AddP5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP5 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1SubP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MulP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP5 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Gcd_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN5 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2AddN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN5 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2SubN5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P2MulN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN5 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddN4 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2AddN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN4 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2SubN4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P2MulN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<P2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN4 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2AddN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN3 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2SubN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P2MulN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN3 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2AddN2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2AddN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN2 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2SubN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MulN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P2PartialDivN2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2AddN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2SubN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MulN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P2DivN1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2DivN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P2PartialDivN1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Add_0 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Add_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Sub_0 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Sub_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Max_0 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Max_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P2Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2AddP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2SubP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MulP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivP1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2DivP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P2PartialDivP1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP1 = <<A as Pow<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2PowP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP1 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<P2CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP2 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2AddP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2SubP2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2SubP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MulP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P2PartialDivP2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP3 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2AddP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2SubP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2MulP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP3 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP3 = <<A as Pow<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2PowP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP4 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P2AddP4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubP4 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2SubP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2MulP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP4 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP5 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2AddP5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP5 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2SubP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P2MulP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP5 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P32 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP5 = <<A as Pow<B>>::Output as Same<P32>>::Output;

    assert_eq!(<P2PowP5 as Integer>::to_i64(), <P32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN5 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3AddN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN5 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3SubN5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN5 = <<A as Mul<B>>::Output as Same<N15>>::Output;
```

```
    assert_eq!(<P3MulN5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN5 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3AddN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN4 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3SubN4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P3MulN4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P3MulN4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN4 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3AddN3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3AddN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN3 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3SubN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3MulN3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<P3MulN3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P3DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P3PartialDivN3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3AddN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN2 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3SubN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P3MulN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3AddN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3SubN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MulN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3DivN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<P3PartialDivN1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Add_0 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Add_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Sub_0 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Sub_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Max_0 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Max_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P3Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP1 = <<A as Add<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P3AddP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3SubP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MulP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3DivP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<P3PartialDivP1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3PowP1 = <<A as Pow<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3PowP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP2 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3AddP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3SubP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3MulP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP3 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3AddP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3SubP3 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P3SubP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3MulP3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P3PartialDivP3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P27 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP3 = <<A as Pow<B>>::Output as Same<P27>>::Output;

    assert_eq!(<P3PowP3 as Integer>::to_i64(), <P27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP4 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3AddP4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3SubP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P3MulP4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP4 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<P3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP5 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3AddP5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP5 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3SubP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP5 = <<A as Mul<B>>::Output as Same<P15>>::Output;
```

```
    assert_eq!(<P3MulP5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP5 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P243 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type P3PowP5 = <<A as Pow<B>>::Output as Same<P243>>::Output;

    assert_eq!(<P3PowP5 as Integer>::to_i64(), <P243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN5 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4AddN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P4SubN5 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4SubN5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P4MulN5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN5 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemN5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4AddN4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4AddN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4SubN4 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4SubN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<P4MulN4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P4PartialDivN4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4AddN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN3 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4SubN3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P4MulN3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P4MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P4MaxN3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddN2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4AddN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN2 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4SubN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P4MulN2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4MaxN2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu
```

```
    assert_eq!(<P4PartialDivN2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4AddN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN1 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4SubN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MulN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4DivN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type P4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<P4PartialDivN1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Add_0 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Add_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Sub_0 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Sub_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Max_0 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Max_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P4Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP1 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4AddP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4SubP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MulP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4DivP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<P4PartialDivP1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PowP1 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4PowP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP2 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P4AddP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubP2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4SubP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4MulP2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P4PartialDivP2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type P4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP3 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4AddP3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4SubP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P4MulP3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P64 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP3 = <<A as Pow<B>>::Output as Same<P64>>::Output;

    assert_eq!(<P4PowP3 as Integer>::to_i64(), <P64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP4 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4AddP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4SubP4 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P4SubP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4MulP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P4PartialDivP4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type P4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<P4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP5 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4AddP5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP5 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4SubP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P4MulP5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP5 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemP5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1024 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P4PowP5 = <<A as Pow<B>>::Output as Same<P1024>>::Output;

    assert_eq!(<P4PowP5 as Integer>::to_i64(), <P1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5AddN5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5AddN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN5 = <<A as Sub<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5SubN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulN5 = <<A as Mul<B>>::Output as Same<N25>>::Output;
```

```
    assert_eq!(<P5MulN5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P5PartialDivN5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5AddN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5SubN4 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5SubN4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulN4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P5MulN4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P5MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5AddN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P5SubN3 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5SubN3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P5MulN3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P5MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5AddN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN2 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5SubN2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulN2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P5MulN2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5AddN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN1 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5SubN1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MulN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5DivN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<P5PartialDivN1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Add_0 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Add_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Sub_0 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Sub_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Max_0 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Max_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P5Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP1 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P5AddP1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5SubP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MulP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5DivP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<P5PartialDivP1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5PowP1 = <<A as Pow<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5PowP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP2 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5AddP2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5SubP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulP2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5MulP2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP3 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5AddP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP3 = <<A as Sub<B>>::Output as Same<P2>>::Output;
```

```
    assert_eq!(<P5SubP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P5MulP3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1
```

```
    #[allow(non_camel_case_types)]
    type P5PowP3 = <<A as Pow<B>>::Output as Same<P125>>::Output;

    assert_eq!(<P5PowP3 as Integer>::to_i64(), <P125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5AddP4 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5AddP4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5SubP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulP4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P5MulP4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type P5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<P5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP5 = <<A as Add<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5AddP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5SubP5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5SubP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulP5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5MulP5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP5 = <<A as Min<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P5MinP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_PartialDiv_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P5PartialDivP5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P5PowP5 = <<A as Pow<B>>::Output as Same<P3125>>::Output;

    assert_eq!(<P5PowP5 as Integer>::to_i64(), <P3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN5 = <<A as Neg>::Output as Same<P5>>::Output;
    assert_eq!(<NegN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN4 = <<A as Neg>::Output as Same<P4>>::Output;
    assert_eq!(<NegN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN3 = <<A as Neg>::Output as Same<P3>>::Output;
    assert_eq!(<NegN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN2 = <<A as Neg>::Output as Same<P2>>::Output;
    assert_eq!(<NegN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Neg() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegN1 = <<A as Neg>::Output as Same<P1>>::Output;
    assert_eq!(<NegN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Abs() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Neg() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Neg_0 = <<A as Neg>::Output as Same<_0>>::Output;
    assert_eq!(<Neg_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Abs() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Abs_0 = <<A as Abs>::Output as Same<_0>>::Output;
    assert_eq!(<Abs_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Neg() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegP1 = <<A as Neg>::Output as Same<N1>>::Output;
    assert_eq!(<NegP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Abs() {
    type A = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP2 = <<A as Neg>::Output as Same<N2>>::Output;
    assert_eq!(<NegP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP3 = <<A as Neg>::Output as Same<N3>>::Output;
    assert_eq!(<NegP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type AbsP3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP4 = <<A as Neg>::Output as Same<N4>>::Output;
    assert_eq!(<NegP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP5 = <<A as Neg>::Output as Same<N5>>::Output;
    assert_eq!(<NegP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

File: ./target/x86_64-pc-windows-gnu/release/build/typenum-10a9d0

```
/**
Convenient type operations.
```

Any types representing values must be able to be expressed as `ident`s. Tha
in scope.

For example, `P5` is okay, but `typenum::P5` is not.

You may combine operators arbitrarily, although doing so excessively may re
recursion limit.

Example
```rust
#![recursion_limit="128"]
#[macro_use] extern crate typenum;
use typenum::consts::*;

fn main() {
    assert_type!(
        op!(min((P1 - P2) * (N3 + N7), P5 * (P3 + P4)) == P10)
    );
}
```
Operators are evaluated based on the operator precedence outlined
[here](https://doc.rust-lang.org/reference.html#operator-precedence).

The full list of supported operators and functions is as follows:

`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, `==`, `!=`, `<=`, `>=`,

They all expand to type aliases defined in the `operator_aliases` module. H
including examples:

Operator `*`. Expands to `Prod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 * P3), P6);
# }
```

Operator `/`. Expands to `Quot`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 / P2), P3);
# }
```

Operator `%`. Expands to `Mod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 % P3), P2);
# }
```

Operator `+`. Expands to `Sum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 + P3), P5);
# }
```

Operator `-`. Expands to `Diff`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 - P3), N1);
# }
```

Operator `<<`. Expands to `Shleft`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U1 << U5), U32);
# }
```

Operator `>>`. Expands to `Shright`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
```

```
assert_type_eq!(op!(U32 >> U5), U1);
# }
```

Operator `&`. Expands to `And`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 & U3), U1);
# }
```

Operator `^`. Expands to `Xor`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 ^ U3), U6);
# }
```

Operator `|`. Expands to `Or`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 | U3), U7);
# }
```

Operator `==`. Expands to `Eq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 == P3 + P2), True);
# }
```

Operator `!=`. Expands to `NotEq`.

```rust
```

```
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 != P3 + P2), False);
# }
```

Operator `<=`. Expands to `LeEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 <= P3 + P2), False);
# }
```

Operator `>=`. Expands to `GrEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 >= P3 + P2), True);
# }
```

Operator `<`. Expands to `Le`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P4 < P3 + P2), True);
# }
```

Operator `>`. Expands to `Gr`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 < P3 + P2), False);
# }
```

Operator `cmp`. Expands to `Compare`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cmp(P2, P3)), Less);
# }
```

Operator `sqr`. Expands to `Square`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqr(P2)), P4);
# }
```

Operator `sqrt`. Expands to `Sqrt`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqrt(U9)), U3);
# }
```

Operator `abs`. Expands to `AbsVal`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(abs(N2)), P2);
# }
```

Operator `cube`. Expands to `Cube`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cube(P2)), P8);
# }
```

```

---
Operator `pow`. Expands to `Exp`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(pow(P2, P3)), P8);
# }
```

Operator `min`. Expands to `Minimum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(min(P2, P3)), P2);
# }
```

Operator `max`. Expands to `Maximum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(max(P2, P3)), P3);
# }
```

Operator `log2`. Expands to `Log2`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(log2(U9)), U3);
# }
```

Operator `gcd`. Expands to `Gcf`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
```
```

```
# fn main() {
assert_type_eq!(op!(gcd(U9, U21)), U3);
# }
```

*/
#[macro_export(local_inner_macros)]
macro_rules! op {
    ($($tail:tt)*) => (__op_internal__!($($tail)*));
}

    #[doc(hidden)]
    #[macro_export(local_inner_macros)]
    macro_rules! __op_internal__ {

(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cmp $($tail:tt)
    __op_internal__!(@stack[Compare, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqr $($tail:tt)
    __op_internal__!(@stack[Square, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqrt $($tail:tt
    __op_internal__!(@stack[Sqrt, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: abs $($tail:tt)
    __op_internal__!(@stack[AbsVal, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cube $($tail:tt
    __op_internal__!(@stack[Cube, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: pow $($tail:tt)
    __op_internal__!(@stack[Exp, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: min $($tail:tt)
    __op_internal__!(@stack[Minimum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: max $($tail:tt)
    __op_internal__!(@stack[Maximum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: log2 $($tail:tt
    __op_internal__!(@stack[Log2, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: gcd $($tail:tt)
    __op_internal__!(@stack[Gcf, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: , $($ta
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail

```
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: *
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: *
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: *
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:tt)*)
    __op_internal__!(@stack[Prod, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: /
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: /
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: /
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:tt)*)
    __op_internal__!(@stack[Quot, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: %
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: %
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: %
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:tt)*)
    __op_internal__!(@stack[Mod, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: +
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: +
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: +
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: +
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: +
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:tt)*)
    __op_internal__!(@stack[Sum, $($stack,)*] @queue[$($queue,)*] @tail: $(
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: -
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: -
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: -
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: -
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: -
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:tt)*)
    __op_internal__!(@stack[Diff, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <<
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <<
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail:tt)*
    __op_internal__!(@stack[Shleft, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >>
);
```

```
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >>
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail:tt)*
    __op_internal__!(@stack[Shright, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: &
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: &
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: &
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: &
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: &
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: &
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:tt)*)
    __op_internal__!(@stack[And, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: ^
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: ^
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ^
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ^
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: ^
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ^
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ^
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:tt)*)
    __op_internal__!(@stack[Xor, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: |
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: |
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: |
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: |
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: |
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: |
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: |
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: | $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:tt)*)
    __op_internal__!(@stack[Or, $($stack,)*] @queue[$($queue,)*] @tail: $($
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: =
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: =
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ==
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ==
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: =
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ==
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ==
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: ==
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: ==
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: =
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: =
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: ==
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: ==
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:tt)*
    __op_internal__!(@stack[Eq, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
```

```
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: !
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: !
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: !=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: !=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: !
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: !=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: !=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: !=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: !=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: !
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: !
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: !=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: !=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:tt)*
    __op_internal__!(@stack[NotEq, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: <=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: <=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: <=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: <=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:tt)*
    __op_internal__!(@stack[LeEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
```

```
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: >=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: >=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: >=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: >=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:tt)*
    __op_internal__!(@stack[GrEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
```

```
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: < $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: < $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: < $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: < $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:tt)*)
    __op_internal__!(@stack[Le, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: > $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: > $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: > $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: > $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:tt)*)
    __op_internal__!(@stack[Gr, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ( $($stuff:tt)*
 => (
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*]
                     @tail: $($stuff)* RParen $($tail)*)
);
```

```
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: RParen
    __op_internal__!(@rp3 @stack[$($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
 => (
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@rp3 @stack[Compare, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Compare, $($queue,)*] @tail
);
(@rp3 @stack[Square, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[Square, $($queue,)*] @tail:
);
(@rp3 @stack[Sqrt, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Sqrt, $($queue,)*] @tail: $
);
(@rp3 @stack[AbsVal, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[AbsVal, $($queue,)*] @tail:
);
(@rp3 @stack[Cube, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Cube, $($queue,)*] @tail: $
);
(@rp3 @stack[Exp, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Exp, $($queue,)*] @tail: $(
);
(@rp3 @stack[Minimum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Minimum, $($queue,)*] @tail
);
(@rp3 @stack[Maximum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Maximum, $($queue,)*] @tail
);
(@rp3 @stack[Log2, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Log2, $($queue,)*] @tail: $
);
(@rp3 @stack[Gcf, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Gcf, $($queue,)*] @tail: $(
);
(@rp3 @stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($tail:tt
    __op_internal__!(@stack[$($stack,)*] @queue[$($queue,)*] @tail: $($tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $num:ident $($t
    __op_internal__!(@stack[$($stack,)*] @queue[$num, $($queue,)*] @tail: $
);
(@stack[] @queue[$($queue:ident,)*] @tail: ) => (
    __op_internal__!(@reverse[] @input: $($queue,)*)
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@reverse[$($revved:ident,)*] @input: $head:ident, $($tail:ident,)* ) => (
    __op_internal__!(@reverse[$head, $($revved,)*] @input: $($tail,)*)
);
```

```
(@reverse[$($revved:ident,)*] @input: ) => (
    __op_internal__!(@eval @stack[] @input[$($revved,)*])
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Prod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Prod<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Quot, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Quot<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Mod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Mod<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Sum, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Sum<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Diff, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Diff<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shleft, $($tail:ident,)*
    __op_internal__!(@eval @stack[$crate::Shleft<$b, $a>, $($stack,)*] @inp
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shright, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Shright<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[And, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::And<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Xor, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Xor<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Or, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Or<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Eq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Eq<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[NotEq, $($tail:ident,)*]
    __op_internal__!(@eval @stack[$crate::NotEq<$b, $a>, $($stack,)*] @inpu
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[LeEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::LeEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[GrEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::GrEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Le, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Le<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gr, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Gr<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Compare, $($tail:ident,)
```

```
    __op_internal__!(@eval @stack[$crate::Compare<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Exp, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Exp<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Minimum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Minimum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Maximum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Maximum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gcf, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Gcf<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Square, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Square<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Sqrt, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Sqrt<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[AbsVal, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::AbsVal<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Cube, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Cube<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Log2, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Log2<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$($stack:ty,)*] @input[$head:ident, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$head, $($stack,)*] @input[$($tail,)*])
);
(@eval @stack[$stack:ty,] @input[]) => (
    $stack
);
($($tail:tt)* ) => (
    __op_internal__!(@stack[] @queue[] @tail: $($tail)*)
);
}
```

File: ./target/x86_64-pc-windows-gnu/release/build/libsqlite3-sys-e3

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
```

```
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
```

```
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
```

```
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
```

```
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
```

```
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
```

```
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
```

```
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
```

```
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
```

```
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
```

```
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
```

```
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xWrite: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *const ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
    >,
    pub xSync: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFileSize: ::std::option::Option<
```

```
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xDeviceCharacteristics: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xShmMap: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        iPg: ::std::os::raw::c_int,
        pgsz: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xShmLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        offset: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
```

```
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
    pub pNext: *mut sqlite3_vfs,
    pub zName: *const ::std::os::raw::c_char,
    pub pAppData: *mut ::std::os::raw::c_void,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: sqlite3_filename,
            arg2: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
            pOutFlags: *mut ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int,
>,
pub xDelete: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        syncDir: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xAccess: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        flags: ::std::os::raw::c_int,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFullPathname: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        nOut: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xDlOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zFilename: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
    >,
    pub xRandomness: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            nByte: ::std::os::raw::c_int,
            zOut: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSleep: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            microseconds: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCurrentTime: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
    >,
    pub xGetLastError: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: ::std::os::raw::c_int,
            arg3: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCurrentTimeInt64: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            arg2: sqlite3_syscall_ptr,
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
```

```
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
```

```
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
```

```
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
```

```
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
```

```
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
```

```
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
```

```
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
```

```
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
```

```
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
```

```
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
```

```
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
```

```
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
```

```
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
```

```
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
```

```
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
```

```
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
```

```
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
```

```
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
```

```
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
```

```
    ) -> ::std::os::raw::c_int,
>,
pub xConnect: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3,
        pAux: *mut ::std::os::raw::c_void,
        argc: ::std::os::raw::c_int,
        argv: *const *const ::std::os::raw::c_char,
        ppVTab: *mut *mut sqlite3_vtab,
        arg2: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xBestIndex: ::std::option::Option<
    unsafe extern "C" fn(
        pVTab: *mut sqlite3_vtab,
        arg1: *mut sqlite3_index_info,
    ) -> ::std::os::raw::c_int,
>,
pub xDisconnect: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xDestroy: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xOpen: ::std::option::Option<
    unsafe extern "C" fn(
        pVTab: *mut sqlite3_vtab,
        ppCursor: *mut *mut sqlite3_vtab_cursor,
    ) -> ::std::os::raw::c_int,
>,
pub xClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xFilter: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        idxNum: ::std::os::raw::c_int,
        idxStr: *const ::std::os::raw::c_char,
        argc: ::std::os::raw::c_int,
        argv: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int,
>,
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
```

```
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        nArg: ::std::os::raw::c_int,
        zName: *const ::std::os::raw::c_char,
        pxFunc: *mut ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        ppArg: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xRename: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        zNew: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSavepoint: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
```

```
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
```

```
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
```

```
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
```

```
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
```

```
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
```

```
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
```

```
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
```

```
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
```

```
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
```

```
pub xUserData: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
>,
pub xColumnCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xRowCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnRow: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnTotalSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xTokenize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pText: *const ::std::os::raw::c_char,
        nText: ::std::os::raw::c_int,
        pCtx: *mut ::std::os::raw::c_void,
        xToken: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
                arg5: ::std::os::raw::c_int,
                arg6: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xPhraseCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xPhraseSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
```

```
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pAux: *mut ::std::os::raw::c_void,
        xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int,
>,
pub xGetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        bClear: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void,
```

```
    >,
    pub xPhraseFirst: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
            arg4: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNext: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
            piOff: *mut ::std::os::raw::c_int,
        ),
    >,
    pub xPhraseFirstColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNextColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
```

```
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./target/x86_64-pc-windows-gnu/debug/build/libsqlite3-sys-aa5

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
```

```
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
```

```
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
```

```
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
```

```
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
```

```
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
```

```
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
```

```
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
```

```
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
```

```
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
```

```
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xWrite: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *const ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
```

```
pub xTruncate: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xDeviceCharacteristics: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xShmMap: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        iPg: ::std::os::raw::c_int,
        pgsz: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
```

```
            arg3: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmLock: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            offset: ::std::os::raw::c_int,
            n: ::std::os::raw::c_int,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
```

```
pub pNext: *mut sqlite3_vfs,
pub zName: *const ::std::os::raw::c_char,
pub pAppData: *mut ::std::os::raw::c_void,
pub xOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: sqlite3_filename,
        arg2: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
        pOutFlags: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xDelete: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        syncDir: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xAccess: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        flags: ::std::os::raw::c_int,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFullPathname: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        nOut: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xDlOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zFilename: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
```

```
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
>,
pub xRandomness: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSleep: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        microseconds: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTime: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
>,
pub xGetLastError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTimeInt64: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xSetSystemCall: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        arg2: sqlite3_syscall_ptr,
    ) -> ::std::os::raw::c_int,
>,
pub xGetSystemCall: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
```

```
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
```

```
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
```

```
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
```

```
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
```

```
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
```

```
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
```

```
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
```

```
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
```

```
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
```

```
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
```

```
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
```

```
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
```

```
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
```

```
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
```

```
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
```

```
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
```

```
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
```

```
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
```

```
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
```

```
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
```

```
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xConnect: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xBestIndex: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: *mut sqlite3_index_info,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDisconnect: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xDestroy: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            ppCursor: *mut *mut sqlite3_vtab_cursor,
        ) -> ::std::os::raw::c_int,
    >,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
    >,
    pub xFilter: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vtab_cursor,
            idxNum: ::std::os::raw::c_int,
            idxStr: *const ::std::os::raw::c_char,
            argc: ::std::os::raw::c_int,
            argv: *mut *mut sqlite3_value,
        ) -> ::std::os::raw::c_int,
    >,
```

```
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        nArg: ::std::os::raw::c_int,
        zName: *const ::std::os::raw::c_char,
        pxFunc: *mut ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        ppArg: *mut *mut ::std::os::raw::c_void,
```

```
        ) -> ::std::os::raw::c_int,
    >,
    pub xRename: ::std::option::Option<
        unsafe extern "C" fn(
            pVtab: *mut sqlite3_vtab,
            zNew: *const ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSavepoint: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
```

```
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
```

```
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
```

```
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
```

```
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
```

```
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
```

```
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
```

```
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
```

```
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
```

```
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
```

```
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
```

```
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
```

```
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
```

```
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
```

```
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
```

```
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
```

```
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
```

```
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
```

```
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
    pub xUserData: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
    >,
    pub xColumnCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xRowCount: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pnRow: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xColumnTotalSize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iCol: ::std::os::raw::c_int,
            pnToken: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            pCtx: *mut ::std::os::raw::c_void,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    arg1: *mut ::std::os::raw::c_void,
                    arg2: ::std::os::raw::c_int,
                    arg3: *const ::std::os::raw::c_char,
                    arg4: ::std::os::raw::c_int,
                    arg5: ::std::os::raw::c_int,
                    arg6: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xPhraseSize: ::std::option::Option<
        unsafe extern "C" fn(
```

```
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            arg1: *mut Fts5Context,
            pAux: *mut ::std::os::raw::c_void,
            xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetAuxdata: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            bClear: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xPhraseFirst: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
            arg4: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNext: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
            piOff: *mut ::std::os::raw::c_int,
        ),
    >,
    pub xPhraseFirstColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNextColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
```

```
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
```

```
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./target/x86_64-pc-windows-gnu/debug/build/typenum-8a82b59

```
/**
Type aliases for many constants.

This file is generated by typenum's build script.

For unsigned integers, the format is `U` followed by the number. We define

- Numbers 0 through 1024
- Powers of 2 below `u64::MAX`
- Powers of 10 below `u64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm};

# #[allow(dead_code)]
type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

For positive signed integers, the format is `P` followed by the number and
signed integers it is `N` followed by the number. For the signed integer ze
`Z0`. We define aliases for

- Numbers -1024 through 1024
- Powers of 2 between `i64::MIN` and `i64::MAX`
- Powers of 10 between `i64::MIN` and `i64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm, PInt, NInt};

# #[allow(dead_code)]
type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
# #[allow(dead_code)]
type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```
```

```
# Example
```rust
# #[allow(unused_imports)]
use typenum::{U0, U1, U2, U3, U4, U5, U6};
# #[allow(unused_imports)]
use typenum::{N3, N2, N1, Z0, P1, P2, P3};
# #[allow(unused_imports)]
use typenum::{U774, N17, N10000, P1024, P4096};
```

We also define the aliases `False` and `True` for `B0` and `B1`, respective
*/
#[allow(missing_docs)]
pub mod consts {
    use crate::uint::{UInt, UTerm};
    use crate::int::{PInt, NInt};

    pub use crate::bit::{B0, B1};
    pub use crate::int::Z0;

    pub type True = B1;
    pub type False = B0;
    pub type U0 = UTerm;
    pub type U1 = UInt<UTerm, B1>;
    pub type P1 = PInt<U1>; pub type N1 = NInt<U1>;
    pub type U2 = UInt<UInt<UTerm, B1>, B0>;
    pub type P2 = PInt<U2>; pub type N2 = NInt<U2>;
    pub type U3 = UInt<UInt<UTerm, B1>, B1>;
    pub type P3 = PInt<U3>; pub type N3 = NInt<U3>;
    pub type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    pub type P4 = PInt<U4>; pub type N4 = NInt<U4>;
    pub type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    pub type P5 = PInt<U5>; pub type N5 = NInt<U5>;
    pub type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
    pub type P6 = PInt<U6>; pub type N6 = NInt<U6>;
    pub type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
    pub type P7 = PInt<U7>; pub type N7 = NInt<U7>;
    pub type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
    pub type P8 = PInt<U8>; pub type N8 = NInt<U8>;
    pub type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;
    pub type P9 = PInt<U9>; pub type N9 = NInt<U9>;
    pub type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;
    pub type P10 = PInt<U10>; pub type N10 = NInt<U10>;
    pub type U11 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>;
    pub type P11 = PInt<U11>; pub type N11 = NInt<U11>;
    pub type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;
    pub type P12 = PInt<U12>; pub type N12 = NInt<U12>;
    pub type U13 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>;
    pub type P13 = PInt<U13>; pub type N13 = NInt<U13>;
    pub type U14 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>;
    pub type P14 = PInt<U14>; pub type N14 = NInt<U14>;
```

```
pub type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;
pub type P15 = PInt<U15>; pub type N15 = NInt<U15>;
pub type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
pub type P16 = PInt<U16>; pub type N16 = NInt<U16>;
pub type U17 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>;
pub type P17 = PInt<U17>; pub type N17 = NInt<U17>;
pub type U18 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B0>;
pub type P18 = PInt<U18>; pub type N18 = NInt<U18>;
pub type U19 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B1>;
pub type P19 = PInt<U19>; pub type N19 = NInt<U19>;
pub type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;
pub type P20 = PInt<U20>; pub type N20 = NInt<U20>;
pub type U21 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B1>;
pub type P21 = PInt<U21>; pub type N21 = NInt<U21>;
pub type U22 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B0>;
pub type P22 = PInt<U22>; pub type N22 = NInt<U22>;
pub type U23 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B1>;
pub type P23 = PInt<U23>; pub type N23 = NInt<U23>;
pub type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;
pub type P24 = PInt<U24>; pub type N24 = NInt<U24>;
pub type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
pub type P25 = PInt<U25>; pub type N25 = NInt<U25>;
pub type U26 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B0>;
pub type P26 = PInt<U26>; pub type N26 = NInt<U26>;
pub type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;
pub type P27 = PInt<U27>; pub type N27 = NInt<U27>;
pub type U28 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B0>;
pub type P28 = PInt<U28>; pub type N28 = NInt<U28>;
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
pub type U35 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P35 = PInt<U35>; pub type N35 = NInt<U35>;
pub type U36 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P36 = PInt<U36>; pub type N36 = NInt<U36>;
pub type U37 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P37 = PInt<U37>; pub type N37 = NInt<U37>;
pub type U38 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P38 = PInt<U38>; pub type N38 = NInt<U38>;
pub type U39 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P39 = PInt<U39>; pub type N39 = NInt<U39>;
pub type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P40 = PInt<U40>; pub type N40 = NInt<U40>;
```

```
pub type U41 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P41 = PInt<U41>; pub type N41 = NInt<U41>;
pub type U42 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P42 = PInt<U42>; pub type N42 = NInt<U42>;
pub type U43 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P43 = PInt<U43>; pub type N43 = NInt<U43>;
pub type U44 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P44 = PInt<U44>; pub type N44 = NInt<U44>;
pub type U45 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P45 = PInt<U45>; pub type N45 = NInt<U45>;
pub type U46 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P46 = PInt<U46>; pub type N46 = NInt<U46>;
pub type U47 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P47 = PInt<U47>; pub type N47 = NInt<U47>;
pub type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P48 = PInt<U48>; pub type N48 = NInt<U48>;
pub type U49 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P49 = PInt<U49>; pub type N49 = NInt<U49>;
pub type U50 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P50 = PInt<U50>; pub type N50 = NInt<U50>;
pub type U51 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P51 = PInt<U51>; pub type N51 = NInt<U51>;
pub type U52 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P52 = PInt<U52>; pub type N52 = NInt<U52>;
pub type U53 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P53 = PInt<U53>; pub type N53 = NInt<U53>;
pub type U54 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P54 = PInt<U54>; pub type N54 = NInt<U54>;
pub type U55 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P55 = PInt<U55>; pub type N55 = NInt<U55>;
pub type U56 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P56 = PInt<U56>; pub type N56 = NInt<U56>;
pub type U57 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P57 = PInt<U57>; pub type N57 = NInt<U57>;
pub type U58 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P58 = PInt<U58>; pub type N58 = NInt<U58>;
pub type U59 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P59 = PInt<U59>; pub type N59 = NInt<U59>;
pub type U60 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P60 = PInt<U60>; pub type N60 = NInt<U60>;
pub type U61 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P61 = PInt<U61>; pub type N61 = NInt<U61>;
pub type U62 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P62 = PInt<U62>; pub type N62 = NInt<U62>;
pub type U63 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P63 = PInt<U63>; pub type N63 = NInt<U63>;
pub type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P64 = PInt<U64>; pub type N64 = NInt<U64>;
pub type U65 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P65 = PInt<U65>; pub type N65 = NInt<U65>;
pub type U66 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P66 = PInt<U66>; pub type N66 = NInt<U66>;
```

```
pub type U67 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P67 = PInt<U67>; pub type N67 = NInt<U67>;
pub type U68 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P68 = PInt<U68>; pub type N68 = NInt<U68>;
pub type U69 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P69 = PInt<U69>; pub type N69 = NInt<U69>;
pub type U70 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P70 = PInt<U70>; pub type N70 = NInt<U70>;
pub type U71 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P71 = PInt<U71>; pub type N71 = NInt<U71>;
pub type U72 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P72 = PInt<U72>; pub type N72 = NInt<U72>;
pub type U73 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P73 = PInt<U73>; pub type N73 = NInt<U73>;
pub type U74 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P74 = PInt<U74>; pub type N74 = NInt<U74>;
pub type U75 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P75 = PInt<U75>; pub type N75 = NInt<U75>;
pub type U76 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P76 = PInt<U76>; pub type N76 = NInt<U76>;
pub type U77 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P77 = PInt<U77>; pub type N77 = NInt<U77>;
pub type U78 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P78 = PInt<U78>; pub type N78 = NInt<U78>;
pub type U79 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P79 = PInt<U79>; pub type N79 = NInt<U79>;
pub type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P80 = PInt<U80>; pub type N80 = NInt<U80>;
pub type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P81 = PInt<U81>; pub type N81 = NInt<U81>;
pub type U82 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P82 = PInt<U82>; pub type N82 = NInt<U82>;
pub type U83 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P83 = PInt<U83>; pub type N83 = NInt<U83>;
pub type U84 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P84 = PInt<U84>; pub type N84 = NInt<U84>;
pub type U85 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P85 = PInt<U85>; pub type N85 = NInt<U85>;
pub type U86 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P86 = PInt<U86>; pub type N86 = NInt<U86>;
pub type U87 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P87 = PInt<U87>; pub type N87 = NInt<U87>;
pub type U88 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P88 = PInt<U88>; pub type N88 = NInt<U88>;
pub type U89 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P89 = PInt<U89>; pub type N89 = NInt<U89>;
pub type U90 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P90 = PInt<U90>; pub type N90 = NInt<U90>;
pub type U91 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P91 = PInt<U91>; pub type N91 = NInt<U91>;
pub type U92 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P92 = PInt<U92>; pub type N92 = NInt<U92>;
```

```
pub type U93 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P93 = PInt<U93>; pub type N93 = NInt<U93>;
pub type U94 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P94 = PInt<U94>; pub type N94 = NInt<U94>;
pub type U95 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P95 = PInt<U95>; pub type N95 = NInt<U95>;
pub type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P96 = PInt<U96>; pub type N96 = NInt<U96>;
pub type U97 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P97 = PInt<U97>; pub type N97 = NInt<U97>;
pub type U98 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P98 = PInt<U98>; pub type N98 = NInt<U98>;
pub type U99 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P99 = PInt<U99>; pub type N99 = NInt<U99>;
pub type U100 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P100 = PInt<U100>; pub type N100 = NInt<U100>;
pub type U101 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P101 = PInt<U101>; pub type N101 = NInt<U101>;
pub type U102 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P102 = PInt<U102>; pub type N102 = NInt<U102>;
pub type U103 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P103 = PInt<U103>; pub type N103 = NInt<U103>;
pub type U104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P104 = PInt<U104>; pub type N104 = NInt<U104>;
pub type U105 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P105 = PInt<U105>; pub type N105 = NInt<U105>;
pub type U106 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P106 = PInt<U106>; pub type N106 = NInt<U106>;
pub type U107 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P107 = PInt<U107>; pub type N107 = NInt<U107>;
pub type U108 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P108 = PInt<U108>; pub type N108 = NInt<U108>;
pub type U109 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P109 = PInt<U109>; pub type N109 = NInt<U109>;
pub type U110 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P110 = PInt<U110>; pub type N110 = NInt<U110>;
pub type U111 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P111 = PInt<U111>; pub type N111 = NInt<U111>;
pub type U112 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P112 = PInt<U112>; pub type N112 = NInt<U112>;
pub type U113 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P113 = PInt<U113>; pub type N113 = NInt<U113>;
pub type U114 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P114 = PInt<U114>; pub type N114 = NInt<U114>;
pub type U115 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P115 = PInt<U115>; pub type N115 = NInt<U115>;
pub type U116 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P116 = PInt<U116>; pub type N116 = NInt<U116>;
pub type U117 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P117 = PInt<U117>; pub type N117 = NInt<U117>;
pub type U118 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P118 = PInt<U118>; pub type N118 = NInt<U118>;
```

```
pub type U119 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P119 = PInt<U119>; pub type N119 = NInt<U119>;
pub type U120 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P120 = PInt<U120>; pub type N120 = NInt<U120>;
pub type U121 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P121 = PInt<U121>; pub type N121 = NInt<U121>;
pub type U122 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P122 = PInt<U122>; pub type N122 = NInt<U122>;
pub type U123 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P123 = PInt<U123>; pub type N123 = NInt<U123>;
pub type U124 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P124 = PInt<U124>; pub type N124 = NInt<U124>;
pub type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P125 = PInt<U125>; pub type N125 = NInt<U125>;
pub type U126 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P126 = PInt<U126>; pub type N126 = NInt<U126>;
pub type U127 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P127 = PInt<U127>; pub type N127 = NInt<U127>;
pub type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P128 = PInt<U128>; pub type N128 = NInt<U128>;
pub type U129 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P129 = PInt<U129>; pub type N129 = NInt<U129>;
pub type U130 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P130 = PInt<U130>; pub type N130 = NInt<U130>;
pub type U131 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P131 = PInt<U131>; pub type N131 = NInt<U131>;
pub type U132 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P132 = PInt<U132>; pub type N132 = NInt<U132>;
pub type U133 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P133 = PInt<U133>; pub type N133 = NInt<U133>;
pub type U134 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P134 = PInt<U134>; pub type N134 = NInt<U134>;
pub type U135 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P135 = PInt<U135>; pub type N135 = NInt<U135>;
pub type U136 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P136 = PInt<U136>; pub type N136 = NInt<U136>;
pub type U137 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P137 = PInt<U137>; pub type N137 = NInt<U137>;
pub type U138 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P138 = PInt<U138>; pub type N138 = NInt<U138>;
pub type U139 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P139 = PInt<U139>; pub type N139 = NInt<U139>;
pub type U140 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P140 = PInt<U140>; pub type N140 = NInt<U140>;
pub type U141 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P141 = PInt<U141>; pub type N141 = NInt<U141>;
pub type U142 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P142 = PInt<U142>; pub type N142 = NInt<U142>;
pub type U143 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P143 = PInt<U143>; pub type N143 = NInt<U143>;
pub type U144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P144 = PInt<U144>; pub type N144 = NInt<U144>;
```

```
pub type U145 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P145 = PInt<U145>; pub type N145 = NInt<U145>;
pub type U146 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P146 = PInt<U146>; pub type N146 = NInt<U146>;
pub type U147 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P147 = PInt<U147>; pub type N147 = NInt<U147>;
pub type U148 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P148 = PInt<U148>; pub type N148 = NInt<U148>;
pub type U149 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P149 = PInt<U149>; pub type N149 = NInt<U149>;
pub type U150 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P150 = PInt<U150>; pub type N150 = NInt<U150>;
pub type U151 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P151 = PInt<U151>; pub type N151 = NInt<U151>;
pub type U152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P152 = PInt<U152>; pub type N152 = NInt<U152>;
pub type U153 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P153 = PInt<U153>; pub type N153 = NInt<U153>;
pub type U154 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P154 = PInt<U154>; pub type N154 = NInt<U154>;
pub type U155 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P155 = PInt<U155>; pub type N155 = NInt<U155>;
pub type U156 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P156 = PInt<U156>; pub type N156 = NInt<U156>;
pub type U157 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P157 = PInt<U157>; pub type N157 = NInt<U157>;
pub type U158 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P158 = PInt<U158>; pub type N158 = NInt<U158>;
pub type U159 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P159 = PInt<U159>; pub type N159 = NInt<U159>;
pub type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P160 = PInt<U160>; pub type N160 = NInt<U160>;
pub type U161 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P161 = PInt<U161>; pub type N161 = NInt<U161>;
pub type U162 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P162 = PInt<U162>; pub type N162 = NInt<U162>;
pub type U163 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P163 = PInt<U163>; pub type N163 = NInt<U163>;
pub type U164 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P164 = PInt<U164>; pub type N164 = NInt<U164>;
pub type U165 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P165 = PInt<U165>; pub type N165 = NInt<U165>;
pub type U166 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P166 = PInt<U166>; pub type N166 = NInt<U166>;
pub type U167 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P167 = PInt<U167>; pub type N167 = NInt<U167>;
pub type U168 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P168 = PInt<U168>; pub type N168 = NInt<U168>;
pub type U169 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P169 = PInt<U169>; pub type N169 = NInt<U169>;
pub type U170 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P170 = PInt<U170>; pub type N170 = NInt<U170>;
```

```
pub type U171 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P171 = PInt<U171>; pub type N171 = NInt<U171>;
pub type U172 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P172 = PInt<U172>; pub type N172 = NInt<U172>;
pub type U173 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P173 = PInt<U173>; pub type N173 = NInt<U173>;
pub type U174 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P174 = PInt<U174>; pub type N174 = NInt<U174>;
pub type U175 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P175 = PInt<U175>; pub type N175 = NInt<U175>;
pub type U176 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P176 = PInt<U176>; pub type N176 = NInt<U176>;
pub type U177 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P177 = PInt<U177>; pub type N177 = NInt<U177>;
pub type U178 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P178 = PInt<U178>; pub type N178 = NInt<U178>;
pub type U179 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P179 = PInt<U179>; pub type N179 = NInt<U179>;
pub type U180 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P180 = PInt<U180>; pub type N180 = NInt<U180>;
pub type U181 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P181 = PInt<U181>; pub type N181 = NInt<U181>;
pub type U182 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P182 = PInt<U182>; pub type N182 = NInt<U182>;
pub type U183 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P183 = PInt<U183>; pub type N183 = NInt<U183>;
pub type U184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P184 = PInt<U184>; pub type N184 = NInt<U184>;
pub type U185 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P185 = PInt<U185>; pub type N185 = NInt<U185>;
pub type U186 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P186 = PInt<U186>; pub type N186 = NInt<U186>;
pub type U187 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P187 = PInt<U187>; pub type N187 = NInt<U187>;
pub type U188 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P188 = PInt<U188>; pub type N188 = NInt<U188>;
pub type U189 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P189 = PInt<U189>; pub type N189 = NInt<U189>;
pub type U190 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P190 = PInt<U190>; pub type N190 = NInt<U190>;
pub type U191 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P191 = PInt<U191>; pub type N191 = NInt<U191>;
pub type U192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P192 = PInt<U192>; pub type N192 = NInt<U192>;
pub type U193 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P193 = PInt<U193>; pub type N193 = NInt<U193>;
pub type U194 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P194 = PInt<U194>; pub type N194 = NInt<U194>;
pub type U195 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P195 = PInt<U195>; pub type N195 = NInt<U195>;
pub type U196 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P196 = PInt<U196>; pub type N196 = NInt<U196>;
```

```
pub type U197 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P197 = PInt<U197>; pub type N197 = NInt<U197>;
pub type U198 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P198 = PInt<U198>; pub type N198 = NInt<U198>;
pub type U199 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P199 = PInt<U199>; pub type N199 = NInt<U199>;
pub type U200 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P200 = PInt<U200>; pub type N200 = NInt<U200>;
pub type U201 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P201 = PInt<U201>; pub type N201 = NInt<U201>;
pub type U202 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P202 = PInt<U202>; pub type N202 = NInt<U202>;
pub type U203 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P203 = PInt<U203>; pub type N203 = NInt<U203>;
pub type U204 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P204 = PInt<U204>; pub type N204 = NInt<U204>;
pub type U205 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P205 = PInt<U205>; pub type N205 = NInt<U205>;
pub type U206 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P206 = PInt<U206>; pub type N206 = NInt<U206>;
pub type U207 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P207 = PInt<U207>; pub type N207 = NInt<U207>;
pub type U208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P208 = PInt<U208>; pub type N208 = NInt<U208>;
pub type U209 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P209 = PInt<U209>; pub type N209 = NInt<U209>;
pub type U210 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P210 = PInt<U210>; pub type N210 = NInt<U210>;
pub type U211 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P211 = PInt<U211>; pub type N211 = NInt<U211>;
pub type U212 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P212 = PInt<U212>; pub type N212 = NInt<U212>;
pub type U213 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P213 = PInt<U213>; pub type N213 = NInt<U213>;
pub type U214 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P214 = PInt<U214>; pub type N214 = NInt<U214>;
pub type U215 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P215 = PInt<U215>; pub type N215 = NInt<U215>;
pub type U216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P216 = PInt<U216>; pub type N216 = NInt<U216>;
pub type U217 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P217 = PInt<U217>; pub type N217 = NInt<U217>;
pub type U218 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P218 = PInt<U218>; pub type N218 = NInt<U218>;
pub type U219 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P219 = PInt<U219>; pub type N219 = NInt<U219>;
pub type U220 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P220 = PInt<U220>; pub type N220 = NInt<U220>;
pub type U221 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P221 = PInt<U221>; pub type N221 = NInt<U221>;
pub type U222 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P222 = PInt<U222>; pub type N222 = NInt<U222>;
```

```
pub type U223 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P223 = PInt<U223>; pub type N223 = NInt<U223>;
pub type U224 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P224 = PInt<U224>; pub type N224 = NInt<U224>;
pub type U225 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P225 = PInt<U225>; pub type N225 = NInt<U225>;
pub type U226 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P226 = PInt<U226>; pub type N226 = NInt<U226>;
pub type U227 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P227 = PInt<U227>; pub type N227 = NInt<U227>;
pub type U228 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P228 = PInt<U228>; pub type N228 = NInt<U228>;
pub type U229 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P229 = PInt<U229>; pub type N229 = NInt<U229>;
pub type U230 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P230 = PInt<U230>; pub type N230 = NInt<U230>;
pub type U231 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P231 = PInt<U231>; pub type N231 = NInt<U231>;
pub type U232 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P232 = PInt<U232>; pub type N232 = NInt<U232>;
pub type U233 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P233 = PInt<U233>; pub type N233 = NInt<U233>;
pub type U234 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P234 = PInt<U234>; pub type N234 = NInt<U234>;
pub type U235 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P235 = PInt<U235>; pub type N235 = NInt<U235>;
pub type U236 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P236 = PInt<U236>; pub type N236 = NInt<U236>;
pub type U237 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P237 = PInt<U237>; pub type N237 = NInt<U237>;
pub type U238 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P238 = PInt<U238>; pub type N238 = NInt<U238>;
pub type U239 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P239 = PInt<U239>; pub type N239 = NInt<U239>;
pub type U240 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P240 = PInt<U240>; pub type N240 = NInt<U240>;
pub type U241 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P241 = PInt<U241>; pub type N241 = NInt<U241>;
pub type U242 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P242 = PInt<U242>; pub type N242 = NInt<U242>;
pub type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P243 = PInt<U243>; pub type N243 = NInt<U243>;
pub type U244 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P244 = PInt<U244>; pub type N244 = NInt<U244>;
pub type U245 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P245 = PInt<U245>; pub type N245 = NInt<U245>;
pub type U246 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P246 = PInt<U246>; pub type N246 = NInt<U246>;
pub type U247 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P247 = PInt<U247>; pub type N247 = NInt<U247>;
pub type U248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P248 = PInt<U248>; pub type N248 = NInt<U248>;
```

```
pub type U249 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P249 = PInt<U249>; pub type N249 = NInt<U249>;
pub type U250 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P250 = PInt<U250>; pub type N250 = NInt<U250>;
pub type U251 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P251 = PInt<U251>; pub type N251 = NInt<U251>;
pub type U252 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P252 = PInt<U252>; pub type N252 = NInt<U252>;
pub type U253 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P253 = PInt<U253>; pub type N253 = NInt<U253>;
pub type U254 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P254 = PInt<U254>; pub type N254 = NInt<U254>;
pub type U255 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P255 = PInt<U255>; pub type N255 = NInt<U255>;
pub type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P256 = PInt<U256>; pub type N256 = NInt<U256>;
pub type U257 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P257 = PInt<U257>; pub type N257 = NInt<U257>;
pub type U258 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P258 = PInt<U258>; pub type N258 = NInt<U258>;
pub type U259 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P259 = PInt<U259>; pub type N259 = NInt<U259>;
pub type U260 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P260 = PInt<U260>; pub type N260 = NInt<U260>;
pub type U261 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P261 = PInt<U261>; pub type N261 = NInt<U261>;
pub type U262 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P262 = PInt<U262>; pub type N262 = NInt<U262>;
pub type U263 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P263 = PInt<U263>; pub type N263 = NInt<U263>;
pub type U264 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P264 = PInt<U264>; pub type N264 = NInt<U264>;
pub type U265 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P265 = PInt<U265>; pub type N265 = NInt<U265>;
pub type U266 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P266 = PInt<U266>; pub type N266 = NInt<U266>;
pub type U267 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P267 = PInt<U267>; pub type N267 = NInt<U267>;
pub type U268 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P268 = PInt<U268>; pub type N268 = NInt<U268>;
pub type U269 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P269 = PInt<U269>; pub type N269 = NInt<U269>;
pub type U270 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P270 = PInt<U270>; pub type N270 = NInt<U270>;
pub type U271 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P271 = PInt<U271>; pub type N271 = NInt<U271>;
pub type U272 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P272 = PInt<U272>; pub type N272 = NInt<U272>;
pub type U273 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P273 = PInt<U273>; pub type N273 = NInt<U273>;
pub type U274 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P274 = PInt<U274>; pub type N274 = NInt<U274>;
```

```
pub type U275 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P275 = PInt<U275>; pub type N275 = NInt<U275>;
pub type U276 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P276 = PInt<U276>; pub type N276 = NInt<U276>;
pub type U277 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P277 = PInt<U277>; pub type N277 = NInt<U277>;
pub type U278 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P278 = PInt<U278>; pub type N278 = NInt<U278>;
pub type U279 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P279 = PInt<U279>; pub type N279 = NInt<U279>;
pub type U280 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P280 = PInt<U280>; pub type N280 = NInt<U280>;
pub type U281 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P281 = PInt<U281>; pub type N281 = NInt<U281>;
pub type U282 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P282 = PInt<U282>; pub type N282 = NInt<U282>;
pub type U283 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P283 = PInt<U283>; pub type N283 = NInt<U283>;
pub type U284 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P284 = PInt<U284>; pub type N284 = NInt<U284>;
pub type U285 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P285 = PInt<U285>; pub type N285 = NInt<U285>;
pub type U286 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P286 = PInt<U286>; pub type N286 = NInt<U286>;
pub type U287 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P287 = PInt<U287>; pub type N287 = NInt<U287>;
pub type U288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P288 = PInt<U288>; pub type N288 = NInt<U288>;
pub type U289 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P289 = PInt<U289>; pub type N289 = NInt<U289>;
pub type U290 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P290 = PInt<U290>; pub type N290 = NInt<U290>;
pub type U291 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P291 = PInt<U291>; pub type N291 = NInt<U291>;
pub type U292 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P292 = PInt<U292>; pub type N292 = NInt<U292>;
pub type U293 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P293 = PInt<U293>; pub type N293 = NInt<U293>;
pub type U294 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P294 = PInt<U294>; pub type N294 = NInt<U294>;
pub type U295 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P295 = PInt<U295>; pub type N295 = NInt<U295>;
pub type U296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P296 = PInt<U296>; pub type N296 = NInt<U296>;
pub type U297 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P297 = PInt<U297>; pub type N297 = NInt<U297>;
pub type U298 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P298 = PInt<U298>; pub type N298 = NInt<U298>;
pub type U299 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P299 = PInt<U299>; pub type N299 = NInt<U299>;
pub type U300 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P300 = PInt<U300>; pub type N300 = NInt<U300>;
```

```
pub type U301 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P301 = PInt<U301>; pub type N301 = NInt<U301>;
pub type U302 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P302 = PInt<U302>; pub type N302 = NInt<U302>;
pub type U303 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P303 = PInt<U303>; pub type N303 = NInt<U303>;
pub type U304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P304 = PInt<U304>; pub type N304 = NInt<U304>;
pub type U305 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P305 = PInt<U305>; pub type N305 = NInt<U305>;
pub type U306 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P306 = PInt<U306>; pub type N306 = NInt<U306>;
pub type U307 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P307 = PInt<U307>; pub type N307 = NInt<U307>;
pub type U308 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P308 = PInt<U308>; pub type N308 = NInt<U308>;
pub type U309 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P309 = PInt<U309>; pub type N309 = NInt<U309>;
pub type U310 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P310 = PInt<U310>; pub type N310 = NInt<U310>;
pub type U311 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P311 = PInt<U311>; pub type N311 = NInt<U311>;
pub type U312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P312 = PInt<U312>; pub type N312 = NInt<U312>;
pub type U313 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P313 = PInt<U313>; pub type N313 = NInt<U313>;
pub type U314 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P314 = PInt<U314>; pub type N314 = NInt<U314>;
pub type U315 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P315 = PInt<U315>; pub type N315 = NInt<U315>;
pub type U316 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P316 = PInt<U316>; pub type N316 = NInt<U316>;
pub type U317 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P317 = PInt<U317>; pub type N317 = NInt<U317>;
pub type U318 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P318 = PInt<U318>; pub type N318 = NInt<U318>;
pub type U319 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P319 = PInt<U319>; pub type N319 = NInt<U319>;
pub type U320 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P320 = PInt<U320>; pub type N320 = NInt<U320>;
pub type U321 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P321 = PInt<U321>; pub type N321 = NInt<U321>;
pub type U322 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P322 = PInt<U322>; pub type N322 = NInt<U322>;
pub type U323 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P323 = PInt<U323>; pub type N323 = NInt<U323>;
pub type U324 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P324 = PInt<U324>; pub type N324 = NInt<U324>;
pub type U325 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P325 = PInt<U325>; pub type N325 = NInt<U325>;
pub type U326 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P326 = PInt<U326>; pub type N326 = NInt<U326>;
```

```
pub type U327 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P327 = PInt<U327>; pub type N327 = NInt<U327>;
pub type U328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P328 = PInt<U328>; pub type N328 = NInt<U328>;
pub type U329 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P329 = PInt<U329>; pub type N329 = NInt<U329>;
pub type U330 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P330 = PInt<U330>; pub type N330 = NInt<U330>;
pub type U331 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P331 = PInt<U331>; pub type N331 = NInt<U331>;
pub type U332 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P332 = PInt<U332>; pub type N332 = NInt<U332>;
pub type U333 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P333 = PInt<U333>; pub type N333 = NInt<U333>;
pub type U334 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P334 = PInt<U334>; pub type N334 = NInt<U334>;
pub type U335 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P335 = PInt<U335>; pub type N335 = NInt<U335>;
pub type U336 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P336 = PInt<U336>; pub type N336 = NInt<U336>;
pub type U337 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P337 = PInt<U337>; pub type N337 = NInt<U337>;
pub type U338 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P338 = PInt<U338>; pub type N338 = NInt<U338>;
pub type U339 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P339 = PInt<U339>; pub type N339 = NInt<U339>;
pub type U340 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P340 = PInt<U340>; pub type N340 = NInt<U340>;
pub type U341 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P341 = PInt<U341>; pub type N341 = NInt<U341>;
pub type U342 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P342 = PInt<U342>; pub type N342 = NInt<U342>;
pub type U343 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P343 = PInt<U343>; pub type N343 = NInt<U343>;
pub type U344 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P344 = PInt<U344>; pub type N344 = NInt<U344>;
pub type U345 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P345 = PInt<U345>; pub type N345 = NInt<U345>;
pub type U346 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P346 = PInt<U346>; pub type N346 = NInt<U346>;
pub type U347 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P347 = PInt<U347>; pub type N347 = NInt<U347>;
pub type U348 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P348 = PInt<U348>; pub type N348 = NInt<U348>;
pub type U349 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P349 = PInt<U349>; pub type N349 = NInt<U349>;
pub type U350 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P350 = PInt<U350>; pub type N350 = NInt<U350>;
pub type U351 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P351 = PInt<U351>; pub type N351 = NInt<U351>;
pub type U352 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P352 = PInt<U352>; pub type N352 = NInt<U352>;
```

```
pub type U353 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P353 = PInt<U353>; pub type N353 = NInt<U353>;
pub type U354 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P354 = PInt<U354>; pub type N354 = NInt<U354>;
pub type U355 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P355 = PInt<U355>; pub type N355 = NInt<U355>;
pub type U356 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P356 = PInt<U356>; pub type N356 = NInt<U356>;
pub type U357 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P357 = PInt<U357>; pub type N357 = NInt<U357>;
pub type U358 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P358 = PInt<U358>; pub type N358 = NInt<U358>;
pub type U359 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P359 = PInt<U359>; pub type N359 = NInt<U359>;
pub type U360 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P360 = PInt<U360>; pub type N360 = NInt<U360>;
pub type U361 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P361 = PInt<U361>; pub type N361 = NInt<U361>;
pub type U362 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P362 = PInt<U362>; pub type N362 = NInt<U362>;
pub type U363 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P363 = PInt<U363>; pub type N363 = NInt<U363>;
pub type U364 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P364 = PInt<U364>; pub type N364 = NInt<U364>;
pub type U365 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P365 = PInt<U365>; pub type N365 = NInt<U365>;
pub type U366 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P366 = PInt<U366>; pub type N366 = NInt<U366>;
pub type U367 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P367 = PInt<U367>; pub type N367 = NInt<U367>;
pub type U368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P368 = PInt<U368>; pub type N368 = NInt<U368>;
pub type U369 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P369 = PInt<U369>; pub type N369 = NInt<U369>;
pub type U370 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P370 = PInt<U370>; pub type N370 = NInt<U370>;
pub type U371 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P371 = PInt<U371>; pub type N371 = NInt<U371>;
pub type U372 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P372 = PInt<U372>; pub type N372 = NInt<U372>;
pub type U373 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P373 = PInt<U373>; pub type N373 = NInt<U373>;
pub type U374 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P374 = PInt<U374>; pub type N374 = NInt<U374>;
pub type U375 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P375 = PInt<U375>; pub type N375 = NInt<U375>;
pub type U376 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P376 = PInt<U376>; pub type N376 = NInt<U376>;
pub type U377 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P377 = PInt<U377>; pub type N377 = NInt<U377>;
pub type U378 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P378 = PInt<U378>; pub type N378 = NInt<U378>;
```

```
pub type U379 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P379 = PInt<U379>; pub type N379 = NInt<U379>;
pub type U380 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P380 = PInt<U380>; pub type N380 = NInt<U380>;
pub type U381 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P381 = PInt<U381>; pub type N381 = NInt<U381>;
pub type U382 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P382 = PInt<U382>; pub type N382 = NInt<U382>;
pub type U383 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P383 = PInt<U383>; pub type N383 = NInt<U383>;
pub type U384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P384 = PInt<U384>; pub type N384 = NInt<U384>;
pub type U385 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P385 = PInt<U385>; pub type N385 = NInt<U385>;
pub type U386 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P386 = PInt<U386>; pub type N386 = NInt<U386>;
pub type U387 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P387 = PInt<U387>; pub type N387 = NInt<U387>;
pub type U388 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P388 = PInt<U388>; pub type N388 = NInt<U388>;
pub type U389 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P389 = PInt<U389>; pub type N389 = NInt<U389>;
pub type U390 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P390 = PInt<U390>; pub type N390 = NInt<U390>;
pub type U391 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P391 = PInt<U391>; pub type N391 = NInt<U391>;
pub type U392 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P392 = PInt<U392>; pub type N392 = NInt<U392>;
pub type U393 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P393 = PInt<U393>; pub type N393 = NInt<U393>;
pub type U394 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P394 = PInt<U394>; pub type N394 = NInt<U394>;
pub type U395 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P395 = PInt<U395>; pub type N395 = NInt<U395>;
pub type U396 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P396 = PInt<U396>; pub type N396 = NInt<U396>;
pub type U397 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P397 = PInt<U397>; pub type N397 = NInt<U397>;
pub type U398 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P398 = PInt<U398>; pub type N398 = NInt<U398>;
pub type U399 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P399 = PInt<U399>; pub type N399 = NInt<U399>;
pub type U400 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P400 = PInt<U400>; pub type N400 = NInt<U400>;
pub type U401 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P401 = PInt<U401>; pub type N401 = NInt<U401>;
pub type U402 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P402 = PInt<U402>; pub type N402 = NInt<U402>;
pub type U403 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P403 = PInt<U403>; pub type N403 = NInt<U403>;
pub type U404 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P404 = PInt<U404>; pub type N404 = NInt<U404>;
```

```
pub type U405 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P405 = PInt<U405>; pub type N405 = NInt<U405>;
pub type U406 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P406 = PInt<U406>; pub type N406 = NInt<U406>;
pub type U407 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P407 = PInt<U407>; pub type N407 = NInt<U407>;
pub type U408 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P408 = PInt<U408>; pub type N408 = NInt<U408>;
pub type U409 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P409 = PInt<U409>; pub type N409 = NInt<U409>;
pub type U410 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P410 = PInt<U410>; pub type N410 = NInt<U410>;
pub type U411 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P411 = PInt<U411>; pub type N411 = NInt<U411>;
pub type U412 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P412 = PInt<U412>; pub type N412 = NInt<U412>;
pub type U413 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P413 = PInt<U413>; pub type N413 = NInt<U413>;
pub type U414 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P414 = PInt<U414>; pub type N414 = NInt<U414>;
pub type U415 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P415 = PInt<U415>; pub type N415 = NInt<U415>;
pub type U416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P416 = PInt<U416>; pub type N416 = NInt<U416>;
pub type U417 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P417 = PInt<U417>; pub type N417 = NInt<U417>;
pub type U418 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P418 = PInt<U418>; pub type N418 = NInt<U418>;
pub type U419 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P419 = PInt<U419>; pub type N419 = NInt<U419>;
pub type U420 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P420 = PInt<U420>; pub type N420 = NInt<U420>;
pub type U421 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P421 = PInt<U421>; pub type N421 = NInt<U421>;
pub type U422 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P422 = PInt<U422>; pub type N422 = NInt<U422>;
pub type U423 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P423 = PInt<U423>; pub type N423 = NInt<U423>;
pub type U424 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P424 = PInt<U424>; pub type N424 = NInt<U424>;
pub type U425 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P425 = PInt<U425>; pub type N425 = NInt<U425>;
pub type U426 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P426 = PInt<U426>; pub type N426 = NInt<U426>;
pub type U427 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P427 = PInt<U427>; pub type N427 = NInt<U427>;
pub type U428 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P428 = PInt<U428>; pub type N428 = NInt<U428>;
pub type U429 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P429 = PInt<U429>; pub type N429 = NInt<U429>;
pub type U430 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P430 = PInt<U430>; pub type N430 = NInt<U430>;
```

```
pub type U431 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P431 = PInt<U431>; pub type N431 = NInt<U431>;
pub type U432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P432 = PInt<U432>; pub type N432 = NInt<U432>;
pub type U433 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P433 = PInt<U433>; pub type N433 = NInt<U433>;
pub type U434 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P434 = PInt<U434>; pub type N434 = NInt<U434>;
pub type U435 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P435 = PInt<U435>; pub type N435 = NInt<U435>;
pub type U436 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P436 = PInt<U436>; pub type N436 = NInt<U436>;
pub type U437 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P437 = PInt<U437>; pub type N437 = NInt<U437>;
pub type U438 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P438 = PInt<U438>; pub type N438 = NInt<U438>;
pub type U439 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P439 = PInt<U439>; pub type N439 = NInt<U439>;
pub type U440 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P440 = PInt<U440>; pub type N440 = NInt<U440>;
pub type U441 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P441 = PInt<U441>; pub type N441 = NInt<U441>;
pub type U442 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P442 = PInt<U442>; pub type N442 = NInt<U442>;
pub type U443 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P443 = PInt<U443>; pub type N443 = NInt<U443>;
pub type U444 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P444 = PInt<U444>; pub type N444 = NInt<U444>;
pub type U445 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P445 = PInt<U445>; pub type N445 = NInt<U445>;
pub type U446 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P446 = PInt<U446>; pub type N446 = NInt<U446>;
pub type U447 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P447 = PInt<U447>; pub type N447 = NInt<U447>;
pub type U448 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P448 = PInt<U448>; pub type N448 = NInt<U448>;
pub type U449 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P449 = PInt<U449>; pub type N449 = NInt<U449>;
pub type U450 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P450 = PInt<U450>; pub type N450 = NInt<U450>;
pub type U451 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P451 = PInt<U451>; pub type N451 = NInt<U451>;
pub type U452 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P452 = PInt<U452>; pub type N452 = NInt<U452>;
pub type U453 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P453 = PInt<U453>; pub type N453 = NInt<U453>;
pub type U454 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P454 = PInt<U454>; pub type N454 = NInt<U454>;
pub type U455 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P455 = PInt<U455>; pub type N455 = NInt<U455>;
pub type U456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P456 = PInt<U456>; pub type N456 = NInt<U456>;
```

```
pub type U457 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P457 = PInt<U457>; pub type N457 = NInt<U457>;
pub type U458 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P458 = PInt<U458>; pub type N458 = NInt<U458>;
pub type U459 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P459 = PInt<U459>; pub type N459 = NInt<U459>;
pub type U460 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P460 = PInt<U460>; pub type N460 = NInt<U460>;
pub type U461 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P461 = PInt<U461>; pub type N461 = NInt<U461>;
pub type U462 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P462 = PInt<U462>; pub type N462 = NInt<U462>;
pub type U463 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P463 = PInt<U463>; pub type N463 = NInt<U463>;
pub type U464 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P464 = PInt<U464>; pub type N464 = NInt<U464>;
pub type U465 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P465 = PInt<U465>; pub type N465 = NInt<U465>;
pub type U466 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P466 = PInt<U466>; pub type N466 = NInt<U466>;
pub type U467 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P467 = PInt<U467>; pub type N467 = NInt<U467>;
pub type U468 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P468 = PInt<U468>; pub type N468 = NInt<U468>;
pub type U469 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P469 = PInt<U469>; pub type N469 = NInt<U469>;
pub type U470 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P470 = PInt<U470>; pub type N470 = NInt<U470>;
pub type U471 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P471 = PInt<U471>; pub type N471 = NInt<U471>;
pub type U472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P472 = PInt<U472>; pub type N472 = NInt<U472>;
pub type U473 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P473 = PInt<U473>; pub type N473 = NInt<U473>;
pub type U474 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P474 = PInt<U474>; pub type N474 = NInt<U474>;
pub type U475 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P475 = PInt<U475>; pub type N475 = NInt<U475>;
pub type U476 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P476 = PInt<U476>; pub type N476 = NInt<U476>;
pub type U477 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P477 = PInt<U477>; pub type N477 = NInt<U477>;
pub type U478 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P478 = PInt<U478>; pub type N478 = NInt<U478>;
pub type U479 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P479 = PInt<U479>; pub type N479 = NInt<U479>;
pub type U480 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P480 = PInt<U480>; pub type N480 = NInt<U480>;
pub type U481 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P481 = PInt<U481>; pub type N481 = NInt<U481>;
pub type U482 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P482 = PInt<U482>; pub type N482 = NInt<U482>;
```

```
pub type U483 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P483 = PInt<U483>; pub type N483 = NInt<U483>;
pub type U484 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P484 = PInt<U484>; pub type N484 = NInt<U484>;
pub type U485 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P485 = PInt<U485>; pub type N485 = NInt<U485>;
pub type U486 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P486 = PInt<U486>; pub type N486 = NInt<U486>;
pub type U487 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P487 = PInt<U487>; pub type N487 = NInt<U487>;
pub type U488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P488 = PInt<U488>; pub type N488 = NInt<U488>;
pub type U489 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P489 = PInt<U489>; pub type N489 = NInt<U489>;
pub type U490 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P490 = PInt<U490>; pub type N490 = NInt<U490>;
pub type U491 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P491 = PInt<U491>; pub type N491 = NInt<U491>;
pub type U492 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P492 = PInt<U492>; pub type N492 = NInt<U492>;
pub type U493 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P493 = PInt<U493>; pub type N493 = NInt<U493>;
pub type U494 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P494 = PInt<U494>; pub type N494 = NInt<U494>;
pub type U495 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P495 = PInt<U495>; pub type N495 = NInt<U495>;
pub type U496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P496 = PInt<U496>; pub type N496 = NInt<U496>;
pub type U497 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P497 = PInt<U497>; pub type N497 = NInt<U497>;
pub type U498 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P498 = PInt<U498>; pub type N498 = NInt<U498>;
pub type U499 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P499 = PInt<U499>; pub type N499 = NInt<U499>;
pub type U500 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P500 = PInt<U500>; pub type N500 = NInt<U500>;
pub type U501 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P501 = PInt<U501>; pub type N501 = NInt<U501>;
pub type U502 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P502 = PInt<U502>; pub type N502 = NInt<U502>;
pub type U503 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P503 = PInt<U503>; pub type N503 = NInt<U503>;
pub type U504 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P504 = PInt<U504>; pub type N504 = NInt<U504>;
pub type U505 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P505 = PInt<U505>; pub type N505 = NInt<U505>;
pub type U506 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P506 = PInt<U506>; pub type N506 = NInt<U506>;
pub type U507 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P507 = PInt<U507>; pub type N507 = NInt<U507>;
pub type U508 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P508 = PInt<U508>; pub type N508 = NInt<U508>;
```

```
pub type U509 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P509 = PInt<U509>; pub type N509 = NInt<U509>;
pub type U510 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P510 = PInt<U510>; pub type N510 = NInt<U510>;
pub type U511 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P511 = PInt<U511>; pub type N511 = NInt<U511>;
pub type U512 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P512 = PInt<U512>; pub type N512 = NInt<U512>;
pub type U513 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P513 = PInt<U513>; pub type N513 = NInt<U513>;
pub type U514 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P514 = PInt<U514>; pub type N514 = NInt<U514>;
pub type U515 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P515 = PInt<U515>; pub type N515 = NInt<U515>;
pub type U516 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P516 = PInt<U516>; pub type N516 = NInt<U516>;
pub type U517 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P517 = PInt<U517>; pub type N517 = NInt<U517>;
pub type U518 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P518 = PInt<U518>; pub type N518 = NInt<U518>;
pub type U519 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P519 = PInt<U519>; pub type N519 = NInt<U519>;
pub type U520 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P520 = PInt<U520>; pub type N520 = NInt<U520>;
pub type U521 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P521 = PInt<U521>; pub type N521 = NInt<U521>;
pub type U522 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P522 = PInt<U522>; pub type N522 = NInt<U522>;
pub type U523 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P523 = PInt<U523>; pub type N523 = NInt<U523>;
pub type U524 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P524 = PInt<U524>; pub type N524 = NInt<U524>;
pub type U525 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P525 = PInt<U525>; pub type N525 = NInt<U525>;
pub type U526 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P526 = PInt<U526>; pub type N526 = NInt<U526>;
pub type U527 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P527 = PInt<U527>; pub type N527 = NInt<U527>;
pub type U528 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P528 = PInt<U528>; pub type N528 = NInt<U528>;
pub type U529 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P529 = PInt<U529>; pub type N529 = NInt<U529>;
pub type U530 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P530 = PInt<U530>; pub type N530 = NInt<U530>;
pub type U531 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P531 = PInt<U531>; pub type N531 = NInt<U531>;
pub type U532 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P532 = PInt<U532>; pub type N532 = NInt<U532>;
pub type U533 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P533 = PInt<U533>; pub type N533 = NInt<U533>;
pub type U534 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P534 = PInt<U534>; pub type N534 = NInt<U534>;
```

```
pub type U535 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P535 = PInt<U535>; pub type N535 = NInt<U535>;
pub type U536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P536 = PInt<U536>; pub type N536 = NInt<U536>;
pub type U537 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P537 = PInt<U537>; pub type N537 = NInt<U537>;
pub type U538 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P538 = PInt<U538>; pub type N538 = NInt<U538>;
pub type U539 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P539 = PInt<U539>; pub type N539 = NInt<U539>;
pub type U540 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P540 = PInt<U540>; pub type N540 = NInt<U540>;
pub type U541 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P541 = PInt<U541>; pub type N541 = NInt<U541>;
pub type U542 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P542 = PInt<U542>; pub type N542 = NInt<U542>;
pub type U543 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P543 = PInt<U543>; pub type N543 = NInt<U543>;
pub type U544 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P544 = PInt<U544>; pub type N544 = NInt<U544>;
pub type U545 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P545 = PInt<U545>; pub type N545 = NInt<U545>;
pub type U546 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P546 = PInt<U546>; pub type N546 = NInt<U546>;
pub type U547 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P547 = PInt<U547>; pub type N547 = NInt<U547>;
pub type U548 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P548 = PInt<U548>; pub type N548 = NInt<U548>;
pub type U549 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P549 = PInt<U549>; pub type N549 = NInt<U549>;
pub type U550 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P550 = PInt<U550>; pub type N550 = NInt<U550>;
pub type U551 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P551 = PInt<U551>; pub type N551 = NInt<U551>;
pub type U552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P552 = PInt<U552>; pub type N552 = NInt<U552>;
pub type U553 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P553 = PInt<U553>; pub type N553 = NInt<U553>;
pub type U554 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P554 = PInt<U554>; pub type N554 = NInt<U554>;
pub type U555 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P555 = PInt<U555>; pub type N555 = NInt<U555>;
pub type U556 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P556 = PInt<U556>; pub type N556 = NInt<U556>;
pub type U557 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P557 = PInt<U557>; pub type N557 = NInt<U557>;
pub type U558 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P558 = PInt<U558>; pub type N558 = NInt<U558>;
pub type U559 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P559 = PInt<U559>; pub type N559 = NInt<U559>;
pub type U560 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P560 = PInt<U560>; pub type N560 = NInt<U560>;
```

```
pub type U561 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P561 = PInt<U561>; pub type N561 = NInt<U561>;
pub type U562 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P562 = PInt<U562>; pub type N562 = NInt<U562>;
pub type U563 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P563 = PInt<U563>; pub type N563 = NInt<U563>;
pub type U564 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P564 = PInt<U564>; pub type N564 = NInt<U564>;
pub type U565 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P565 = PInt<U565>; pub type N565 = NInt<U565>;
pub type U566 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P566 = PInt<U566>; pub type N566 = NInt<U566>;
pub type U567 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P567 = PInt<U567>; pub type N567 = NInt<U567>;
pub type U568 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P568 = PInt<U568>; pub type N568 = NInt<U568>;
pub type U569 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P569 = PInt<U569>; pub type N569 = NInt<U569>;
pub type U570 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P570 = PInt<U570>; pub type N570 = NInt<U570>;
pub type U571 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P571 = PInt<U571>; pub type N571 = NInt<U571>;
pub type U572 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P572 = PInt<U572>; pub type N572 = NInt<U572>;
pub type U573 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P573 = PInt<U573>; pub type N573 = NInt<U573>;
pub type U574 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P574 = PInt<U574>; pub type N574 = NInt<U574>;
pub type U575 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P575 = PInt<U575>; pub type N575 = NInt<U575>;
pub type U576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P576 = PInt<U576>; pub type N576 = NInt<U576>;
pub type U577 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P577 = PInt<U577>; pub type N577 = NInt<U577>;
pub type U578 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P578 = PInt<U578>; pub type N578 = NInt<U578>;
pub type U579 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P579 = PInt<U579>; pub type N579 = NInt<U579>;
pub type U580 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P580 = PInt<U580>; pub type N580 = NInt<U580>;
pub type U581 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P581 = PInt<U581>; pub type N581 = NInt<U581>;
pub type U582 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P582 = PInt<U582>; pub type N582 = NInt<U582>;
pub type U583 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P583 = PInt<U583>; pub type N583 = NInt<U583>;
pub type U584 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P584 = PInt<U584>; pub type N584 = NInt<U584>;
pub type U585 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P585 = PInt<U585>; pub type N585 = NInt<U585>;
pub type U586 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P586 = PInt<U586>; pub type N586 = NInt<U586>;
```

```
pub type U587 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P587 = PInt<U587>; pub type N587 = NInt<U587>;
pub type U588 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P588 = PInt<U588>; pub type N588 = NInt<U588>;
pub type U589 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P589 = PInt<U589>; pub type N589 = NInt<U589>;
pub type U590 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P590 = PInt<U590>; pub type N590 = NInt<U590>;
pub type U591 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P591 = PInt<U591>; pub type N591 = NInt<U591>;
pub type U592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P592 = PInt<U592>; pub type N592 = NInt<U592>;
pub type U593 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P593 = PInt<U593>; pub type N593 = NInt<U593>;
pub type U594 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P594 = PInt<U594>; pub type N594 = NInt<U594>;
pub type U595 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P595 = PInt<U595>; pub type N595 = NInt<U595>;
pub type U596 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P596 = PInt<U596>; pub type N596 = NInt<U596>;
pub type U597 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P597 = PInt<U597>; pub type N597 = NInt<U597>;
pub type U598 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P598 = PInt<U598>; pub type N598 = NInt<U598>;
pub type U599 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P599 = PInt<U599>; pub type N599 = NInt<U599>;
pub type U600 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P600 = PInt<U600>; pub type N600 = NInt<U600>;
pub type U601 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P601 = PInt<U601>; pub type N601 = NInt<U601>;
pub type U602 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P602 = PInt<U602>; pub type N602 = NInt<U602>;
pub type U603 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P603 = PInt<U603>; pub type N603 = NInt<U603>;
pub type U604 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P604 = PInt<U604>; pub type N604 = NInt<U604>;
pub type U605 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P605 = PInt<U605>; pub type N605 = NInt<U605>;
pub type U606 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P606 = PInt<U606>; pub type N606 = NInt<U606>;
pub type U607 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P607 = PInt<U607>; pub type N607 = NInt<U607>;
pub type U608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P608 = PInt<U608>; pub type N608 = NInt<U608>;
pub type U609 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P609 = PInt<U609>; pub type N609 = NInt<U609>;
pub type U610 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P610 = PInt<U610>; pub type N610 = NInt<U610>;
pub type U611 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P611 = PInt<U611>; pub type N611 = NInt<U611>;
pub type U612 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P612 = PInt<U612>; pub type N612 = NInt<U612>;
```

```
pub type U613 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P613 = PInt<U613>; pub type N613 = NInt<U613>;
pub type U614 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P614 = PInt<U614>; pub type N614 = NInt<U614>;
pub type U615 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P615 = PInt<U615>; pub type N615 = NInt<U615>;
pub type U616 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P616 = PInt<U616>; pub type N616 = NInt<U616>;
pub type U617 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P617 = PInt<U617>; pub type N617 = NInt<U617>;
pub type U618 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P618 = PInt<U618>; pub type N618 = NInt<U618>;
pub type U619 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P619 = PInt<U619>; pub type N619 = NInt<U619>;
pub type U620 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P620 = PInt<U620>; pub type N620 = NInt<U620>;
pub type U621 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P621 = PInt<U621>; pub type N621 = NInt<U621>;
pub type U622 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P622 = PInt<U622>; pub type N622 = NInt<U622>;
pub type U623 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P623 = PInt<U623>; pub type N623 = NInt<U623>;
pub type U624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P624 = PInt<U624>; pub type N624 = NInt<U624>;
pub type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P625 = PInt<U625>; pub type N625 = NInt<U625>;
pub type U626 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P626 = PInt<U626>; pub type N626 = NInt<U626>;
pub type U627 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P627 = PInt<U627>; pub type N627 = NInt<U627>;
pub type U628 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P628 = PInt<U628>; pub type N628 = NInt<U628>;
pub type U629 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P629 = PInt<U629>; pub type N629 = NInt<U629>;
pub type U630 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P630 = PInt<U630>; pub type N630 = NInt<U630>;
pub type U631 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P631 = PInt<U631>; pub type N631 = NInt<U631>;
pub type U632 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P632 = PInt<U632>; pub type N632 = NInt<U632>;
pub type U633 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P633 = PInt<U633>; pub type N633 = NInt<U633>;
pub type U634 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P634 = PInt<U634>; pub type N634 = NInt<U634>;
pub type U635 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P635 = PInt<U635>; pub type N635 = NInt<U635>;
pub type U636 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P636 = PInt<U636>; pub type N636 = NInt<U636>;
pub type U637 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P637 = PInt<U637>; pub type N637 = NInt<U637>;
pub type U638 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P638 = PInt<U638>; pub type N638 = NInt<U638>;
```

```
pub type U639 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P639 = PInt<U639>; pub type N639 = NInt<U639>;
pub type U640 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P640 = PInt<U640>; pub type N640 = NInt<U640>;
pub type U641 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P641 = PInt<U641>; pub type N641 = NInt<U641>;
pub type U642 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P642 = PInt<U642>; pub type N642 = NInt<U642>;
pub type U643 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P643 = PInt<U643>; pub type N643 = NInt<U643>;
pub type U644 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P644 = PInt<U644>; pub type N644 = NInt<U644>;
pub type U645 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P645 = PInt<U645>; pub type N645 = NInt<U645>;
pub type U646 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P646 = PInt<U646>; pub type N646 = NInt<U646>;
pub type U647 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P647 = PInt<U647>; pub type N647 = NInt<U647>;
pub type U648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P648 = PInt<U648>; pub type N648 = NInt<U648>;
pub type U649 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P649 = PInt<U649>; pub type N649 = NInt<U649>;
pub type U650 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P650 = PInt<U650>; pub type N650 = NInt<U650>;
pub type U651 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P651 = PInt<U651>; pub type N651 = NInt<U651>;
pub type U652 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P652 = PInt<U652>; pub type N652 = NInt<U652>;
pub type U653 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P653 = PInt<U653>; pub type N653 = NInt<U653>;
pub type U654 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P654 = PInt<U654>; pub type N654 = NInt<U654>;
pub type U655 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P655 = PInt<U655>; pub type N655 = NInt<U655>;
pub type U656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P656 = PInt<U656>; pub type N656 = NInt<U656>;
pub type U657 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P657 = PInt<U657>; pub type N657 = NInt<U657>;
pub type U658 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P658 = PInt<U658>; pub type N658 = NInt<U658>;
pub type U659 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P659 = PInt<U659>; pub type N659 = NInt<U659>;
pub type U660 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P660 = PInt<U660>; pub type N660 = NInt<U660>;
pub type U661 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P661 = PInt<U661>; pub type N661 = NInt<U661>;
pub type U662 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P662 = PInt<U662>; pub type N662 = NInt<U662>;
pub type U663 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P663 = PInt<U663>; pub type N663 = NInt<U663>;
pub type U664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P664 = PInt<U664>; pub type N664 = NInt<U664>;
```

```
pub type U665 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P665 = PInt<U665>; pub type N665 = NInt<U665>;
pub type U666 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P666 = PInt<U666>; pub type N666 = NInt<U666>;
pub type U667 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P667 = PInt<U667>; pub type N667 = NInt<U667>;
pub type U668 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P668 = PInt<U668>; pub type N668 = NInt<U668>;
pub type U669 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P669 = PInt<U669>; pub type N669 = NInt<U669>;
pub type U670 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P670 = PInt<U670>; pub type N670 = NInt<U670>;
pub type U671 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P671 = PInt<U671>; pub type N671 = NInt<U671>;
pub type U672 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P672 = PInt<U672>; pub type N672 = NInt<U672>;
pub type U673 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P673 = PInt<U673>; pub type N673 = NInt<U673>;
pub type U674 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P674 = PInt<U674>; pub type N674 = NInt<U674>;
pub type U675 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P675 = PInt<U675>; pub type N675 = NInt<U675>;
pub type U676 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P676 = PInt<U676>; pub type N676 = NInt<U676>;
pub type U677 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P677 = PInt<U677>; pub type N677 = NInt<U677>;
pub type U678 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P678 = PInt<U678>; pub type N678 = NInt<U678>;
pub type U679 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P679 = PInt<U679>; pub type N679 = NInt<U679>;
pub type U680 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P680 = PInt<U680>; pub type N680 = NInt<U680>;
pub type U681 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P681 = PInt<U681>; pub type N681 = NInt<U681>;
pub type U682 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P682 = PInt<U682>; pub type N682 = NInt<U682>;
pub type U683 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P683 = PInt<U683>; pub type N683 = NInt<U683>;
pub type U684 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P684 = PInt<U684>; pub type N684 = NInt<U684>;
pub type U685 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P685 = PInt<U685>; pub type N685 = NInt<U685>;
pub type U686 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P686 = PInt<U686>; pub type N686 = NInt<U686>;
pub type U687 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P687 = PInt<U687>; pub type N687 = NInt<U687>;
pub type U688 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P688 = PInt<U688>; pub type N688 = NInt<U688>;
pub type U689 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P689 = PInt<U689>; pub type N689 = NInt<U689>;
pub type U690 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P690 = PInt<U690>; pub type N690 = NInt<U690>;
```

```
pub type U691 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P691 = PInt<U691>; pub type N691 = NInt<U691>;
pub type U692 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P692 = PInt<U692>; pub type N692 = NInt<U692>;
pub type U693 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P693 = PInt<U693>; pub type N693 = NInt<U693>;
pub type U694 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P694 = PInt<U694>; pub type N694 = NInt<U694>;
pub type U695 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P695 = PInt<U695>; pub type N695 = NInt<U695>;
pub type U696 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P696 = PInt<U696>; pub type N696 = NInt<U696>;
pub type U697 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P697 = PInt<U697>; pub type N697 = NInt<U697>;
pub type U698 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P698 = PInt<U698>; pub type N698 = NInt<U698>;
pub type U699 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P699 = PInt<U699>; pub type N699 = NInt<U699>;
pub type U700 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P700 = PInt<U700>; pub type N700 = NInt<U700>;
pub type U701 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P701 = PInt<U701>; pub type N701 = NInt<U701>;
pub type U702 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P702 = PInt<U702>; pub type N702 = NInt<U702>;
pub type U703 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P703 = PInt<U703>; pub type N703 = NInt<U703>;
pub type U704 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P704 = PInt<U704>; pub type N704 = NInt<U704>;
pub type U705 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P705 = PInt<U705>; pub type N705 = NInt<U705>;
pub type U706 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P706 = PInt<U706>; pub type N706 = NInt<U706>;
pub type U707 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P707 = PInt<U707>; pub type N707 = NInt<U707>;
pub type U708 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P708 = PInt<U708>; pub type N708 = NInt<U708>;
pub type U709 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P709 = PInt<U709>; pub type N709 = NInt<U709>;
pub type U710 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P710 = PInt<U710>; pub type N710 = NInt<U710>;
pub type U711 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P711 = PInt<U711>; pub type N711 = NInt<U711>;
pub type U712 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P712 = PInt<U712>; pub type N712 = NInt<U712>;
pub type U713 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P713 = PInt<U713>; pub type N713 = NInt<U713>;
pub type U714 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P714 = PInt<U714>; pub type N714 = NInt<U714>;
pub type U715 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P715 = PInt<U715>; pub type N715 = NInt<U715>;
pub type U716 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P716 = PInt<U716>; pub type N716 = NInt<U716>;
```

```
pub type U717 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P717 = PInt<U717>; pub type N717 = NInt<U717>;
pub type U718 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P718 = PInt<U718>; pub type N718 = NInt<U718>;
pub type U719 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P719 = PInt<U719>; pub type N719 = NInt<U719>;
pub type U720 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P720 = PInt<U720>; pub type N720 = NInt<U720>;
pub type U721 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P721 = PInt<U721>; pub type N721 = NInt<U721>;
pub type U722 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P722 = PInt<U722>; pub type N722 = NInt<U722>;
pub type U723 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P723 = PInt<U723>; pub type N723 = NInt<U723>;
pub type U724 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P724 = PInt<U724>; pub type N724 = NInt<U724>;
pub type U725 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P725 = PInt<U725>; pub type N725 = NInt<U725>;
pub type U726 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P726 = PInt<U726>; pub type N726 = NInt<U726>;
pub type U727 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P727 = PInt<U727>; pub type N727 = NInt<U727>;
pub type U728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P728 = PInt<U728>; pub type N728 = NInt<U728>;
pub type U729 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P729 = PInt<U729>; pub type N729 = NInt<U729>;
pub type U730 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P730 = PInt<U730>; pub type N730 = NInt<U730>;
pub type U731 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P731 = PInt<U731>; pub type N731 = NInt<U731>;
pub type U732 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P732 = PInt<U732>; pub type N732 = NInt<U732>;
pub type U733 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P733 = PInt<U733>; pub type N733 = NInt<U733>;
pub type U734 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P734 = PInt<U734>; pub type N734 = NInt<U734>;
pub type U735 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P735 = PInt<U735>; pub type N735 = NInt<U735>;
pub type U736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P736 = PInt<U736>; pub type N736 = NInt<U736>;
pub type U737 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P737 = PInt<U737>; pub type N737 = NInt<U737>;
pub type U738 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P738 = PInt<U738>; pub type N738 = NInt<U738>;
pub type U739 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P739 = PInt<U739>; pub type N739 = NInt<U739>;
pub type U740 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P740 = PInt<U740>; pub type N740 = NInt<U740>;
pub type U741 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P741 = PInt<U741>; pub type N741 = NInt<U741>;
pub type U742 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P742 = PInt<U742>; pub type N742 = NInt<U742>;
```

```
pub type U743 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P743 = PInt<U743>; pub type N743 = NInt<U743>;
pub type U744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P744 = PInt<U744>; pub type N744 = NInt<U744>;
pub type U745 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P745 = PInt<U745>; pub type N745 = NInt<U745>;
pub type U746 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P746 = PInt<U746>; pub type N746 = NInt<U746>;
pub type U747 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P747 = PInt<U747>; pub type N747 = NInt<U747>;
pub type U748 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P748 = PInt<U748>; pub type N748 = NInt<U748>;
pub type U749 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P749 = PInt<U749>; pub type N749 = NInt<U749>;
pub type U750 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P750 = PInt<U750>; pub type N750 = NInt<U750>;
pub type U751 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P751 = PInt<U751>; pub type N751 = NInt<U751>;
pub type U752 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P752 = PInt<U752>; pub type N752 = NInt<U752>;
pub type U753 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P753 = PInt<U753>; pub type N753 = NInt<U753>;
pub type U754 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P754 = PInt<U754>; pub type N754 = NInt<U754>;
pub type U755 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P755 = PInt<U755>; pub type N755 = NInt<U755>;
pub type U756 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P756 = PInt<U756>; pub type N756 = NInt<U756>;
pub type U757 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P757 = PInt<U757>; pub type N757 = NInt<U757>;
pub type U758 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P758 = PInt<U758>; pub type N758 = NInt<U758>;
pub type U759 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P759 = PInt<U759>; pub type N759 = NInt<U759>;
pub type U760 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P760 = PInt<U760>; pub type N760 = NInt<U760>;
pub type U761 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P761 = PInt<U761>; pub type N761 = NInt<U761>;
pub type U762 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P762 = PInt<U762>; pub type N762 = NInt<U762>;
pub type U763 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P763 = PInt<U763>; pub type N763 = NInt<U763>;
pub type U764 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P764 = PInt<U764>; pub type N764 = NInt<U764>;
pub type U765 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P765 = PInt<U765>; pub type N765 = NInt<U765>;
pub type U766 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P766 = PInt<U766>; pub type N766 = NInt<U766>;
pub type U767 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P767 = PInt<U767>; pub type N767 = NInt<U767>;
pub type U768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P768 = PInt<U768>; pub type N768 = NInt<U768>;
```

```
pub type U769 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P769 = PInt<U769>; pub type N769 = NInt<U769>;
pub type U770 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P770 = PInt<U770>; pub type N770 = NInt<U770>;
pub type U771 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P771 = PInt<U771>; pub type N771 = NInt<U771>;
pub type U772 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P772 = PInt<U772>; pub type N772 = NInt<U772>;
pub type U773 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P773 = PInt<U773>; pub type N773 = NInt<U773>;
pub type U774 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P774 = PInt<U774>; pub type N774 = NInt<U774>;
pub type U775 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P775 = PInt<U775>; pub type N775 = NInt<U775>;
pub type U776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P776 = PInt<U776>; pub type N776 = NInt<U776>;
pub type U777 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P777 = PInt<U777>; pub type N777 = NInt<U777>;
pub type U778 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P778 = PInt<U778>; pub type N778 = NInt<U778>;
pub type U779 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P779 = PInt<U779>; pub type N779 = NInt<U779>;
pub type U780 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P780 = PInt<U780>; pub type N780 = NInt<U780>;
pub type U781 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P781 = PInt<U781>; pub type N781 = NInt<U781>;
pub type U782 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P782 = PInt<U782>; pub type N782 = NInt<U782>;
pub type U783 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P783 = PInt<U783>; pub type N783 = NInt<U783>;
pub type U784 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P784 = PInt<U784>; pub type N784 = NInt<U784>;
pub type U785 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P785 = PInt<U785>; pub type N785 = NInt<U785>;
pub type U786 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P786 = PInt<U786>; pub type N786 = NInt<U786>;
pub type U787 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P787 = PInt<U787>; pub type N787 = NInt<U787>;
pub type U788 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P788 = PInt<U788>; pub type N788 = NInt<U788>;
pub type U789 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P789 = PInt<U789>; pub type N789 = NInt<U789>;
pub type U790 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P790 = PInt<U790>; pub type N790 = NInt<U790>;
pub type U791 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P791 = PInt<U791>; pub type N791 = NInt<U791>;
pub type U792 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P792 = PInt<U792>; pub type N792 = NInt<U792>;
pub type U793 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P793 = PInt<U793>; pub type N793 = NInt<U793>;
pub type U794 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P794 = PInt<U794>; pub type N794 = NInt<U794>;
```

```
pub type U795 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P795 = PInt<U795>; pub type N795 = NInt<U795>;
pub type U796 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P796 = PInt<U796>; pub type N796 = NInt<U796>;
pub type U797 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P797 = PInt<U797>; pub type N797 = NInt<U797>;
pub type U798 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P798 = PInt<U798>; pub type N798 = NInt<U798>;
pub type U799 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P799 = PInt<U799>; pub type N799 = NInt<U799>;
pub type U800 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P800 = PInt<U800>; pub type N800 = NInt<U800>;
pub type U801 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P801 = PInt<U801>; pub type N801 = NInt<U801>;
pub type U802 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P802 = PInt<U802>; pub type N802 = NInt<U802>;
pub type U803 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P803 = PInt<U803>; pub type N803 = NInt<U803>;
pub type U804 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P804 = PInt<U804>; pub type N804 = NInt<U804>;
pub type U805 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P805 = PInt<U805>; pub type N805 = NInt<U805>;
pub type U806 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P806 = PInt<U806>; pub type N806 = NInt<U806>;
pub type U807 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P807 = PInt<U807>; pub type N807 = NInt<U807>;
pub type U808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P808 = PInt<U808>; pub type N808 = NInt<U808>;
pub type U809 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P809 = PInt<U809>; pub type N809 = NInt<U809>;
pub type U810 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P810 = PInt<U810>; pub type N810 = NInt<U810>;
pub type U811 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P811 = PInt<U811>; pub type N811 = NInt<U811>;
pub type U812 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P812 = PInt<U812>; pub type N812 = NInt<U812>;
pub type U813 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P813 = PInt<U813>; pub type N813 = NInt<U813>;
pub type U814 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P814 = PInt<U814>; pub type N814 = NInt<U814>;
pub type U815 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P815 = PInt<U815>; pub type N815 = NInt<U815>;
pub type U816 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P816 = PInt<U816>; pub type N816 = NInt<U816>;
pub type U817 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P817 = PInt<U817>; pub type N817 = NInt<U817>;
pub type U818 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P818 = PInt<U818>; pub type N818 = NInt<U818>;
pub type U819 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P819 = PInt<U819>; pub type N819 = NInt<U819>;
pub type U820 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P820 = PInt<U820>; pub type N820 = NInt<U820>;
```

```
pub type U821 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P821 = PInt<U821>; pub type N821 = NInt<U821>;
pub type U822 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P822 = PInt<U822>; pub type N822 = NInt<U822>;
pub type U823 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P823 = PInt<U823>; pub type N823 = NInt<U823>;
pub type U824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P824 = PInt<U824>; pub type N824 = NInt<U824>;
pub type U825 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P825 = PInt<U825>; pub type N825 = NInt<U825>;
pub type U826 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P826 = PInt<U826>; pub type N826 = NInt<U826>;
pub type U827 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P827 = PInt<U827>; pub type N827 = NInt<U827>;
pub type U828 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P828 = PInt<U828>; pub type N828 = NInt<U828>;
pub type U829 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P829 = PInt<U829>; pub type N829 = NInt<U829>;
pub type U830 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P830 = PInt<U830>; pub type N830 = NInt<U830>;
pub type U831 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P831 = PInt<U831>; pub type N831 = NInt<U831>;
pub type U832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P832 = PInt<U832>; pub type N832 = NInt<U832>;
pub type U833 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P833 = PInt<U833>; pub type N833 = NInt<U833>;
pub type U834 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P834 = PInt<U834>; pub type N834 = NInt<U834>;
pub type U835 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P835 = PInt<U835>; pub type N835 = NInt<U835>;
pub type U836 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P836 = PInt<U836>; pub type N836 = NInt<U836>;
pub type U837 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P837 = PInt<U837>; pub type N837 = NInt<U837>;
pub type U838 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P838 = PInt<U838>; pub type N838 = NInt<U838>;
pub type U839 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P839 = PInt<U839>; pub type N839 = NInt<U839>;
pub type U840 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P840 = PInt<U840>; pub type N840 = NInt<U840>;
pub type U841 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P841 = PInt<U841>; pub type N841 = NInt<U841>;
pub type U842 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P842 = PInt<U842>; pub type N842 = NInt<U842>;
pub type U843 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P843 = PInt<U843>; pub type N843 = NInt<U843>;
pub type U844 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P844 = PInt<U844>; pub type N844 = NInt<U844>;
pub type U845 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P845 = PInt<U845>; pub type N845 = NInt<U845>;
pub type U846 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P846 = PInt<U846>; pub type N846 = NInt<U846>;
```

```
pub type U847 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P847 = PInt<U847>; pub type N847 = NInt<U847>;
pub type U848 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P848 = PInt<U848>; pub type N848 = NInt<U848>;
pub type U849 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P849 = PInt<U849>; pub type N849 = NInt<U849>;
pub type U850 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P850 = PInt<U850>; pub type N850 = NInt<U850>;
pub type U851 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P851 = PInt<U851>; pub type N851 = NInt<U851>;
pub type U852 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P852 = PInt<U852>; pub type N852 = NInt<U852>;
pub type U853 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P853 = PInt<U853>; pub type N853 = NInt<U853>;
pub type U854 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P854 = PInt<U854>; pub type N854 = NInt<U854>;
pub type U855 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P855 = PInt<U855>; pub type N855 = NInt<U855>;
pub type U856 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P856 = PInt<U856>; pub type N856 = NInt<U856>;
pub type U857 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P857 = PInt<U857>; pub type N857 = NInt<U857>;
pub type U858 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P858 = PInt<U858>; pub type N858 = NInt<U858>;
pub type U859 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P859 = PInt<U859>; pub type N859 = NInt<U859>;
pub type U860 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P860 = PInt<U860>; pub type N860 = NInt<U860>;
pub type U861 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P861 = PInt<U861>; pub type N861 = NInt<U861>;
pub type U862 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P862 = PInt<U862>; pub type N862 = NInt<U862>;
pub type U863 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P863 = PInt<U863>; pub type N863 = NInt<U863>;
pub type U864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P864 = PInt<U864>; pub type N864 = NInt<U864>;
pub type U865 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P865 = PInt<U865>; pub type N865 = NInt<U865>;
pub type U866 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P866 = PInt<U866>; pub type N866 = NInt<U866>;
pub type U867 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P867 = PInt<U867>; pub type N867 = NInt<U867>;
pub type U868 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P868 = PInt<U868>; pub type N868 = NInt<U868>;
pub type U869 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P869 = PInt<U869>; pub type N869 = NInt<U869>;
pub type U870 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P870 = PInt<U870>; pub type N870 = NInt<U870>;
pub type U871 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P871 = PInt<U871>; pub type N871 = NInt<U871>;
pub type U872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P872 = PInt<U872>; pub type N872 = NInt<U872>;
```

```
pub type U873 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P873 = PInt<U873>; pub type N873 = NInt<U873>;
pub type U874 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P874 = PInt<U874>; pub type N874 = NInt<U874>;
pub type U875 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P875 = PInt<U875>; pub type N875 = NInt<U875>;
pub type U876 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P876 = PInt<U876>; pub type N876 = NInt<U876>;
pub type U877 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P877 = PInt<U877>; pub type N877 = NInt<U877>;
pub type U878 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P878 = PInt<U878>; pub type N878 = NInt<U878>;
pub type U879 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P879 = PInt<U879>; pub type N879 = NInt<U879>;
pub type U880 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P880 = PInt<U880>; pub type N880 = NInt<U880>;
pub type U881 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P881 = PInt<U881>; pub type N881 = NInt<U881>;
pub type U882 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P882 = PInt<U882>; pub type N882 = NInt<U882>;
pub type U883 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P883 = PInt<U883>; pub type N883 = NInt<U883>;
pub type U884 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P884 = PInt<U884>; pub type N884 = NInt<U884>;
pub type U885 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P885 = PInt<U885>; pub type N885 = NInt<U885>;
pub type U886 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P886 = PInt<U886>; pub type N886 = NInt<U886>;
pub type U887 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P887 = PInt<U887>; pub type N887 = NInt<U887>;
pub type U888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P888 = PInt<U888>; pub type N888 = NInt<U888>;
pub type U889 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P889 = PInt<U889>; pub type N889 = NInt<U889>;
pub type U890 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P890 = PInt<U890>; pub type N890 = NInt<U890>;
pub type U891 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P891 = PInt<U891>; pub type N891 = NInt<U891>;
pub type U892 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P892 = PInt<U892>; pub type N892 = NInt<U892>;
pub type U893 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P893 = PInt<U893>; pub type N893 = NInt<U893>;
pub type U894 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P894 = PInt<U894>; pub type N894 = NInt<U894>;
pub type U895 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P895 = PInt<U895>; pub type N895 = NInt<U895>;
pub type U896 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P896 = PInt<U896>; pub type N896 = NInt<U896>;
pub type U897 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P897 = PInt<U897>; pub type N897 = NInt<U897>;
pub type U898 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P898 = PInt<U898>; pub type N898 = NInt<U898>;
```

```
pub type U899 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P899 = PInt<U899>; pub type N899 = NInt<U899>;
pub type U900 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P900 = PInt<U900>; pub type N900 = NInt<U900>;
pub type U901 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P901 = PInt<U901>; pub type N901 = NInt<U901>;
pub type U902 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P902 = PInt<U902>; pub type N902 = NInt<U902>;
pub type U903 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P903 = PInt<U903>; pub type N903 = NInt<U903>;
pub type U904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P904 = PInt<U904>; pub type N904 = NInt<U904>;
pub type U905 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P905 = PInt<U905>; pub type N905 = NInt<U905>;
pub type U906 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P906 = PInt<U906>; pub type N906 = NInt<U906>;
pub type U907 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P907 = PInt<U907>; pub type N907 = NInt<U907>;
pub type U908 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P908 = PInt<U908>; pub type N908 = NInt<U908>;
pub type U909 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P909 = PInt<U909>; pub type N909 = NInt<U909>;
pub type U910 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P910 = PInt<U910>; pub type N910 = NInt<U910>;
pub type U911 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P911 = PInt<U911>; pub type N911 = NInt<U911>;
pub type U912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P912 = PInt<U912>; pub type N912 = NInt<U912>;
pub type U913 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P913 = PInt<U913>; pub type N913 = NInt<U913>;
pub type U914 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P914 = PInt<U914>; pub type N914 = NInt<U914>;
pub type U915 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P915 = PInt<U915>; pub type N915 = NInt<U915>;
pub type U916 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P916 = PInt<U916>; pub type N916 = NInt<U916>;
pub type U917 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P917 = PInt<U917>; pub type N917 = NInt<U917>;
pub type U918 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P918 = PInt<U918>; pub type N918 = NInt<U918>;
pub type U919 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P919 = PInt<U919>; pub type N919 = NInt<U919>;
pub type U920 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P920 = PInt<U920>; pub type N920 = NInt<U920>;
pub type U921 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P921 = PInt<U921>; pub type N921 = NInt<U921>;
pub type U922 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P922 = PInt<U922>; pub type N922 = NInt<U922>;
pub type U923 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P923 = PInt<U923>; pub type N923 = NInt<U923>;
pub type U924 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P924 = PInt<U924>; pub type N924 = NInt<U924>;
```

```
pub type U925 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P925 = PInt<U925>; pub type N925 = NInt<U925>;
pub type U926 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P926 = PInt<U926>; pub type N926 = NInt<U926>;
pub type U927 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P927 = PInt<U927>; pub type N927 = NInt<U927>;
pub type U928 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P928 = PInt<U928>; pub type N928 = NInt<U928>;
pub type U929 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P929 = PInt<U929>; pub type N929 = NInt<U929>;
pub type U930 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P930 = PInt<U930>; pub type N930 = NInt<U930>;
pub type U931 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P931 = PInt<U931>; pub type N931 = NInt<U931>;
pub type U932 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P932 = PInt<U932>; pub type N932 = NInt<U932>;
pub type U933 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P933 = PInt<U933>; pub type N933 = NInt<U933>;
pub type U934 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P934 = PInt<U934>; pub type N934 = NInt<U934>;
pub type U935 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P935 = PInt<U935>; pub type N935 = NInt<U935>;
pub type U936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P936 = PInt<U936>; pub type N936 = NInt<U936>;
pub type U937 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P937 = PInt<U937>; pub type N937 = NInt<U937>;
pub type U938 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P938 = PInt<U938>; pub type N938 = NInt<U938>;
pub type U939 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P939 = PInt<U939>; pub type N939 = NInt<U939>;
pub type U940 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P940 = PInt<U940>; pub type N940 = NInt<U940>;
pub type U941 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P941 = PInt<U941>; pub type N941 = NInt<U941>;
pub type U942 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P942 = PInt<U942>; pub type N942 = NInt<U942>;
pub type U943 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P943 = PInt<U943>; pub type N943 = NInt<U943>;
pub type U944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P944 = PInt<U944>; pub type N944 = NInt<U944>;
pub type U945 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P945 = PInt<U945>; pub type N945 = NInt<U945>;
pub type U946 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P946 = PInt<U946>; pub type N946 = NInt<U946>;
pub type U947 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P947 = PInt<U947>; pub type N947 = NInt<U947>;
pub type U948 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P948 = PInt<U948>; pub type N948 = NInt<U948>;
pub type U949 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P949 = PInt<U949>; pub type N949 = NInt<U949>;
pub type U950 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P950 = PInt<U950>; pub type N950 = NInt<U950>;
```

```
pub type U951 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P951 = PInt<U951>; pub type N951 = NInt<U951>;
pub type U952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P952 = PInt<U952>; pub type N952 = NInt<U952>;
pub type U953 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P953 = PInt<U953>; pub type N953 = NInt<U953>;
pub type U954 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P954 = PInt<U954>; pub type N954 = NInt<U954>;
pub type U955 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P955 = PInt<U955>; pub type N955 = NInt<U955>;
pub type U956 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P956 = PInt<U956>; pub type N956 = NInt<U956>;
pub type U957 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P957 = PInt<U957>; pub type N957 = NInt<U957>;
pub type U958 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P958 = PInt<U958>; pub type N958 = NInt<U958>;
pub type U959 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P959 = PInt<U959>; pub type N959 = NInt<U959>;
pub type U960 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P960 = PInt<U960>; pub type N960 = NInt<U960>;
pub type U961 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P961 = PInt<U961>; pub type N961 = NInt<U961>;
pub type U962 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P962 = PInt<U962>; pub type N962 = NInt<U962>;
pub type U963 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P963 = PInt<U963>; pub type N963 = NInt<U963>;
pub type U964 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P964 = PInt<U964>; pub type N964 = NInt<U964>;
pub type U965 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P965 = PInt<U965>; pub type N965 = NInt<U965>;
pub type U966 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P966 = PInt<U966>; pub type N966 = NInt<U966>;
pub type U967 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P967 = PInt<U967>; pub type N967 = NInt<U967>;
pub type U968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P968 = PInt<U968>; pub type N968 = NInt<U968>;
pub type U969 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P969 = PInt<U969>; pub type N969 = NInt<U969>;
pub type U970 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P970 = PInt<U970>; pub type N970 = NInt<U970>;
pub type U971 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P971 = PInt<U971>; pub type N971 = NInt<U971>;
pub type U972 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P972 = PInt<U972>; pub type N972 = NInt<U972>;
pub type U973 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P973 = PInt<U973>; pub type N973 = NInt<U973>;
pub type U974 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P974 = PInt<U974>; pub type N974 = NInt<U974>;
pub type U975 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P975 = PInt<U975>; pub type N975 = NInt<U975>;
pub type U976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P976 = PInt<U976>; pub type N976 = NInt<U976>;
```

```
pub type U977 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P977 = PInt<U977>; pub type N977 = NInt<U977>;
pub type U978 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P978 = PInt<U978>; pub type N978 = NInt<U978>;
pub type U979 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P979 = PInt<U979>; pub type N979 = NInt<U979>;
pub type U980 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P980 = PInt<U980>; pub type N980 = NInt<U980>;
pub type U981 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P981 = PInt<U981>; pub type N981 = NInt<U981>;
pub type U982 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P982 = PInt<U982>; pub type N982 = NInt<U982>;
pub type U983 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P983 = PInt<U983>; pub type N983 = NInt<U983>;
pub type U984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P984 = PInt<U984>; pub type N984 = NInt<U984>;
pub type U985 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P985 = PInt<U985>; pub type N985 = NInt<U985>;
pub type U986 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P986 = PInt<U986>; pub type N986 = NInt<U986>;
pub type U987 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P987 = PInt<U987>; pub type N987 = NInt<U987>;
pub type U988 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P988 = PInt<U988>; pub type N988 = NInt<U988>;
pub type U989 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P989 = PInt<U989>; pub type N989 = NInt<U989>;
pub type U990 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P990 = PInt<U990>; pub type N990 = NInt<U990>;
pub type U991 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P991 = PInt<U991>; pub type N991 = NInt<U991>;
pub type U992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P992 = PInt<U992>; pub type N992 = NInt<U992>;
pub type U993 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P993 = PInt<U993>; pub type N993 = NInt<U993>;
pub type U994 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P994 = PInt<U994>; pub type N994 = NInt<U994>;
pub type U995 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P995 = PInt<U995>; pub type N995 = NInt<U995>;
pub type U996 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P996 = PInt<U996>; pub type N996 = NInt<U996>;
pub type U997 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P997 = PInt<U997>; pub type N997 = NInt<U997>;
pub type U998 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P998 = PInt<U998>; pub type N998 = NInt<U998>;
pub type U999 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P999 = PInt<U999>; pub type N999 = NInt<U999>;
pub type U1000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1000 = PInt<U1000>; pub type N1000 = NInt<U1000>;
pub type U1001 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1001 = PInt<U1001>; pub type N1001 = NInt<U1001>;
pub type U1002 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1002 = PInt<U1002>; pub type N1002 = NInt<U1002>;
```

```
pub type U1003 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1003 = PInt<U1003>; pub type N1003 = NInt<U1003>;
pub type U1004 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1004 = PInt<U1004>; pub type N1004 = NInt<U1004>;
pub type U1005 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1005 = PInt<U1005>; pub type N1005 = NInt<U1005>;
pub type U1006 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1006 = PInt<U1006>; pub type N1006 = NInt<U1006>;
pub type U1007 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1007 = PInt<U1007>; pub type N1007 = NInt<U1007>;
pub type U1008 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1008 = PInt<U1008>; pub type N1008 = NInt<U1008>;
pub type U1009 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1009 = PInt<U1009>; pub type N1009 = NInt<U1009>;
pub type U1010 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1010 = PInt<U1010>; pub type N1010 = NInt<U1010>;
pub type U1011 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1011 = PInt<U1011>; pub type N1011 = NInt<U1011>;
pub type U1012 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1012 = PInt<U1012>; pub type N1012 = NInt<U1012>;
pub type U1013 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1013 = PInt<U1013>; pub type N1013 = NInt<U1013>;
pub type U1014 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1014 = PInt<U1014>; pub type N1014 = NInt<U1014>;
pub type U1015 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1015 = PInt<U1015>; pub type N1015 = NInt<U1015>;
pub type U1016 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1016 = PInt<U1016>; pub type N1016 = NInt<U1016>;
pub type U1017 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1017 = PInt<U1017>; pub type N1017 = NInt<U1017>;
pub type U1018 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1018 = PInt<U1018>; pub type N1018 = NInt<U1018>;
pub type U1019 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1019 = PInt<U1019>; pub type N1019 = NInt<U1019>;
pub type U1020 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1020 = PInt<U1020>; pub type N1020 = NInt<U1020>;
pub type U1021 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1021 = PInt<U1021>; pub type N1021 = NInt<U1021>;
pub type U1022 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1022 = PInt<U1022>; pub type N1022 = NInt<U1022>;
pub type U1023 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1023 = PInt<U1023>; pub type N1023 = NInt<U1023>;
pub type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1024 = PInt<U1024>; pub type N1024 = NInt<U1024>;
pub type U2048 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2048 = PInt<U2048>; pub type N2048 = NInt<U2048>;
pub type U4096 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4096 = PInt<U4096>; pub type N4096 = NInt<U4096>;
pub type U8192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P8192 = PInt<U8192>; pub type N8192 = NInt<U8192>;
pub type U16384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P16384 = PInt<U16384>; pub type N16384 = NInt<U16384>;
```

```
pub type U32768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P32768 = PInt<U32768>; pub type N32768 = NInt<U32768>;
pub type U65536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P65536 = PInt<U65536>; pub type N65536 = NInt<U65536>;
pub type U131072 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P131072 = PInt<U131072>; pub type N131072 = NInt<U131072>;
pub type U262144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P262144 = PInt<U262144>; pub type N262144 = NInt<U262144>;
pub type U524288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P524288 = PInt<U524288>; pub type N524288 = NInt<U524288>;
pub type U1048576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1048576 = PInt<U1048576>; pub type N1048576 = NInt<U1048576>;
pub type U2097152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P2097152 = PInt<U2097152>; pub type N2097152 = NInt<U2097152>;
pub type U4194304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P4194304 = PInt<U4194304>; pub type N4194304 = NInt<U4194304>;
pub type U8388608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P8388608 = PInt<U8388608>; pub type N8388608 = NInt<U8388608>;
pub type U16777216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P16777216 = PInt<U16777216>; pub type N16777216 = NInt<U167772
pub type U33554432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P33554432 = PInt<U33554432>; pub type N33554432 = NInt<U335544
pub type U67108864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P67108864 = PInt<U67108864>; pub type N67108864 = NInt<U671088
pub type U134217728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P134217728 = PInt<U134217728>; pub type N134217728 = NInt<U134
pub type U268435456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P268435456 = PInt<U268435456>; pub type N268435456 = NInt<U268
pub type U536870912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P536870912 = PInt<U536870912>; pub type N536870912 = NInt<U536
pub type U1073741824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1073741824 = PInt<U1073741824>; pub type N1073741824 = NInt<U
pub type U2147483648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P2147483648 = PInt<U2147483648>; pub type N2147483648 = NInt<U
pub type U4294967296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P4294967296 = PInt<U4294967296>; pub type N4294967296 = NInt<U
pub type U8589934592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P8589934592 = PInt<U8589934592>; pub type N8589934592 = NInt<U
pub type U17179869184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P17179869184 = PInt<U17179869184>; pub type N17179869184 = NIn
pub type U34359738368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P34359738368 = PInt<U34359738368>; pub type N34359738368 = NIn
pub type U68719476736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P68719476736 = PInt<U68719476736>; pub type N68719476736 = NIn
pub type U137438953472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P137438953472 = PInt<U137438953472>; pub type N137438953472 =
pub type U274877906944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P274877906944 = PInt<U274877906944>; pub type N274877906944 =
pub type U549755813888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P549755813888 = PInt<U549755813888>; pub type N549755813888 =
pub type U1099511627776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1099511627776 = PInt<U1099511627776>; pub type N1099511627776
```

```
pub type U2199023255552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P2199023255552 = PInt<U2199023255552>; pub type N2199023255552
pub type U4398046511104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P4398046511104 = PInt<U4398046511104>; pub type N4398046511104
pub type U8796093022208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P8796093022208 = PInt<U8796093022208>; pub type N8796093022208
pub type U17592186044416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P17592186044416 = PInt<U17592186044416>; pub type N1759218604
pub type U35184372088832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P35184372088832 = PInt<U35184372088832>; pub type N3518437208
pub type U70368744177664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P70368744177664 = PInt<U70368744177664>; pub type N7036874417
pub type U140737488355328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P140737488355328 = PInt<U140737488355328>; pub type N14073748
pub type U281474976710656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P281474976710656 = PInt<U281474976710656>; pub type N28147497
pub type U562949953421312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P562949953421312 = PInt<U562949953421312>; pub type N56294995
pub type U1125899906842624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P1125899906842624 = PInt<U1125899906842624>; pub type N112589
pub type U2251799813685248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P2251799813685248 = PInt<U2251799813685248>; pub type N225179
pub type U4503599627370496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P4503599627370496 = PInt<U4503599627370496>; pub type N450359
pub type U9007199254740992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P9007199254740992 = PInt<U9007199254740992>; pub type N900719
pub type U18014398509481984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P18014398509481984 = PInt<U18014398509481984>; pub type N18014
pub type U36028797018963968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P36028797018963968 = PInt<U36028797018963968>; pub type N36028
pub type U72057594037927936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P72057594037927936 = PInt<U72057594037927936>; pub type N72057
pub type U144115188075855872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P144115188075855872 = PInt<U144115188075855872>; pub type N144
pub type U288230376151711744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P288230376151711744 = PInt<U288230376151711744>; pub type N288
pub type U576460752303423488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P576460752303423488 = PInt<U576460752303423488>; pub type N576
pub type U1152921504606846976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1152921504606846976 = PInt<U1152921504606846976>; pub type N1
pub type U2305843009213693952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2305843009213693952 = PInt<U2305843009213693952>; pub type N2
pub type U4611686018427387904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4611686018427387904 = PInt<U4611686018427387904>; pub type N4
pub type U9223372036854775808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type U10000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P10000 = PInt<U10000>; pub type N10000 = NInt<U10000>;
pub type U100000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P100000 = PInt<U100000>; pub type N100000 = NInt<U100000>;
pub type U1000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1000000 = PInt<U1000000>; pub type N1000000 = NInt<U1000000>;
pub type U10000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
```

```
    pub type P10000000 = PInt<U10000000>; pub type N10000000 = NInt<U100000
    pub type U100000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P100000000 = PInt<U100000000>; pub type N100000000 = NInt<U100
    pub type U1000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P1000000000 = PInt<U1000000000>; pub type N1000000000 = NInt<U
    pub type U10000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P10000000000 = PInt<U10000000000>; pub type N10000000000 = NIn
    pub type U100000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P100000000000 = PInt<U100000000000>; pub type N100000000000 =
    pub type U1000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P1000000000000 = PInt<U1000000000000>; pub type N1000000000000
    pub type U10000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P10000000000000 = PInt<U10000000000000>; pub type N10000000000
    pub type U100000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P100000000000000 = PInt<U100000000000000>; pub type N100000000
    pub type U1000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P1000000000000000 = PInt<U1000000000000000>; pub type N1000000
    pub type U10000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P10000000000000000 = PInt<U10000000000000000>; pub type N10000
    pub type U100000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P100000000000000000 = PInt<U100000000000000000>; pub type N100
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P1000000000000000000 = PInt<U1000000000000000000>; pub type N1
    pub type U10000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
}
```

File: ./target/x86_64-pc-windows-gnu/debug/build/typenum-8a82b59

```
extern crate typenum;

use std::ops::*;
use std::cmp::Ordering;
use typenum::*;

#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_0() {
```

```
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitOrU0 = <<A as BitOr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitOrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitXorU0 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitXorU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU0 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU0 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0AddU0 = <<A as Add<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0AddU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MaxU0 = <<A as Max<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MaxU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0GcdU0 = <<A as Gcd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0GcdU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Sub_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0SubU0 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0SubU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_0() {
    type A = UTerm;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_0() {
    type A = UTerm;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U0CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU1 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitXorU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU1 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU1 = <<A as Add<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U0AddU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU1 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU1 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_Div_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU1 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU1 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U0CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU2 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitXorU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU2 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU2 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0AddU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU2 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_2() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU2 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU2 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_BitXor_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU3 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitXorU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU3 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU3 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0AddU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU3 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU3 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU3 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_4() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU4 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitXorU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU4 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU4 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0AddU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU4 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU4 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_4() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU4 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU5 = <<A as BitXor<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U0BitXorU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU5 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU5 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0AddU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU5 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_Max_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU5 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU5 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU0 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU0 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitXorU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShlU0 = <<A as Shl<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShlU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShrU0 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_0() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU0 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1AddU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU0 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU0 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U1SubU0 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1SubU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U1CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_BitOr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitXorU1 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitXorU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU1 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1ShlU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
```

```
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU1 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1AddU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1SubU1 = <<A as Sub<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U1SubU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU1 = <<A as Mul<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MulU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1DivU1 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1DivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_PartialDiv_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U1PartialDivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Pow_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU1 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U1BitXorU2 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitXorU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU2 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1ShlU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU2 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1AddU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU2 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU2 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MulU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_2() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU2 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;
```

```
    assert_eq!(<U1BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU3 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1BitXorU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU3 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U1ShlU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU3 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1AddU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Min_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU3 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU3 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MulU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU3 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU4 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitXorU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU4 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U1ShlU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_4() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU4 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1AddU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU4 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U1MulU4 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MulU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU4 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_BitAnd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU5 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1BitXorU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U1ShlU5 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U1ShlU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU5 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U1AddU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU5 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U1GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU5 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MulU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU5 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU5 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Cmp_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU0 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU0 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitXorU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2ShlU0 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShlU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShrU0 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU0 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2AddU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU0 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU0 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2SubU0 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2SubU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_0() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U2CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU1 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitXorU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU1 = <<A as Shl<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U2ShlU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU1 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2AddU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU1 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2SubU1 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2SubU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU1 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MulU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2DivU1 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2DivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U2PartialDivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU1 = <<A as Pow<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2PowU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitXorU2 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitXorU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU2 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2ShlU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_2() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU2 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2AddU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U2SubU2 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2SubU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU2 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MulU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U2PartialDivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU2 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2PowU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU3 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU3 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2BitXorU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU3 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2ShlU3 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU3 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2AddU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU3 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2MinU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU3 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2MulU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU3 = <<A as Pow<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2PowU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2BitOrU4 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitOrU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU4 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitXorU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU4 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2ShlU4 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU4 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2AddU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU4 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU4 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU4 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2MulU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_4() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU4 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU4 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2PowU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U2BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU5 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitXorU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU5 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U2ShlU5 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU5 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2AddU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU5 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU5 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U2MulU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU5 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2PowU5 = <<A as Pow<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2PowU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU5 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU0 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU0 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitXorU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShlU0 = <<A as Shl<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShlU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_0() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU0 = <<A as Shr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU0 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3AddU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU0 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3GcdU0 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU0 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3SubU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U3CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU1 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitXorU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU1 = <<A as Shl<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3ShlU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU1 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3AddU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU1 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U3GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3SubU1 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3SubU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU1 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MulU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU1 = <<A as Div<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3DivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U3>>::Outpu

    assert_eq!(<U3PartialDivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU1 = <<A as Pow<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3PowU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU2 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitXorU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU2 = <<A as Shl<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3ShlU2 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU2 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3AddU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU2 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU2 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3SubU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_2() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU2 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3MulU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU2 = <<A as Pow<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3PowU2 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU3 = <<A as BitAnd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitAndU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitXorU3 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitXorU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU3 = <<A as Shl<B>>::Output as Same<U24>>::Output;

    assert_eq!(<U3ShlU3 as Unsigned>::to_u64(), <U24 as Unsigned>::to_u64()
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU3 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3AddU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3SubU3 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3SubU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU3 = <<A as Mul<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3MulU3 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U3PartialDivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U27 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU3 = <<A as Pow<B>>::Output as Same<U27>>::Output;

    assert_eq!(<U3PowU3 as Unsigned>::to_u64(), <U27 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_4() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU4 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU4 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitXorU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U48 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU4 = <<A as Shl<B>>::Output as Same<U48>>::Output;

    assert_eq!(<U3ShlU4 as Unsigned>::to_u64(), <U48 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3AddU4 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3AddU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU4 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU4 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3MulU4 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU4 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U3PowU4 = <<A as Pow<B>>::Output as Same<U81>>::Output;

    assert_eq!(<U3PowU4 as Unsigned>::to_u64(), <U81 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU5 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3BitXorU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU5 = <<A as Shl<B>>::Output as Same<U96>>::Output;

    assert_eq!(<U3ShlU5 as Unsigned>::to_u64(), <U96 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU5 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U3AddU5 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU5 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU5 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U3MulU5 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU5 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U3PowU5 = <<A as Pow<B>>::Output as Same<U243>>::Output;

    assert_eq!(<U3PowU5 as Unsigned>::to_u64(), <U243 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU0 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU0 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitXorU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU0 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShlU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU0 = <<A as Shr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU0 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4AddU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU0 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_0() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU0 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU0 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4SubU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U4CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU1 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitXorU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU1 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4ShlU1 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU1 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4AddU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU1 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU1 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4SubU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU1 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MulU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU1 = <<A as Div<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4DivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U4RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U4>>::Outpu

    assert_eq!(<U4PartialDivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU1 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4PowU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_2() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU2 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitOrU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU2 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitXorU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU2 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4ShlU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4AddU2 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4AddU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU2 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU2 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4SubU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU2 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4MulU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U4PartialDivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_2() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU2 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4PowU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU3 = <<A as BitXor<B>>::Output as Same<U7>>::Output;
```

```
    assert_eq!(<U4BitXorU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU3 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U4ShlU3 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU3 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4AddU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU3 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU3 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4SubU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U12 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU3 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U4MulU3 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4PowU3 = <<A as Pow<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4PowU3 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitXorU4 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitXorU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU4 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4ShlU4 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_4() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU4 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4AddU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4SubU4 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4SubU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU4 = <<A as Mul<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4MulU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U4PartialDivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0

    #[allow(non_camel_case_types)]
    type U4PowU4 = <<A as Pow<B>>::Output as Same<U256>>::Output;

    assert_eq!(<U4PowU4 as Unsigned>::to_u64(), <U256 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU5 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU5 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4BitXorU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0

    #[allow(non_camel_case_types)]
    type U4ShlU5 = <<A as Shl<B>>::Output as Same<U128>>::Output;

    assert_eq!(<U4ShlU5 as Unsigned>::to_u64(), <U128 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU5 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U4AddU5 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU5 = <<A as Min<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4MinU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU5 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U4MulU5 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4RemU5 = <<A as Rem<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4RemU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTe

    #[allow(non_camel_case_types)]
    type U4PowU5 = <<A as Pow<B>>::Output as Same<U1024>>::Output;

    assert_eq!(<U4PowU5 as Unsigned>::to_u64(), <U1024 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitOrU0 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU0 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitXorU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShlU0 = <<A as Shl<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShlU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU0 = <<A as Shr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU0 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5AddU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU0 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU0 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU0 = <<A as Sub<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5SubU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_0() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U5CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U5BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU1 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitXorU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU1 = <<A as Shl<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5ShlU1 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU1 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5AddU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU1 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU1 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5SubU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
```

```
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU1 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MulU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU1 = <<A as Div<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5DivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U5>>::Outpu

    assert_eq!(<U5PartialDivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU1 = <<A as Pow<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U5PowU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU2 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU2 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitXorU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_2() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU2 = <<A as Shl<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5ShlU2 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU2 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5AddU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5MaxU2 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU2 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5SubU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU2 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5MulU2 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU2 = <<A as Pow<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5PowU2 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU3 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5BitXorU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU3 = <<A as Shl<B>>::Output as Same<U40>>::Output;

    assert_eq!(<U5ShlU3 as Unsigned>::to_u64(), <U40 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU3 = <<A as Add<B>>::Output as Same<U8>>::Output;
```

```
    assert_eq!(<U5AddU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU3 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU3 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5SubU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU3 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U5MulU3 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U5PowU3 = <<A as Pow<B>>::Output as Same<U125>>::Output;

    assert_eq!(<U5PowU3 as Unsigned>::to_u64(), <U125 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU4 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitXorU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU4 = <<A as Shl<B>>::Output as Same<U80>>::Output;

    assert_eq!(<U5ShlU4 as Unsigned>::to_u64(), <U80 as Unsigned>::to_u64()
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU4 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U5AddU4 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU4 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_4() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU4 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5SubU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU4 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5MulU4 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type U5PowU4 = <<A as Pow<B>>::Output as Same<U625>>::Output;

    assert_eq!(<U5PowU4 as Unsigned>::to_u64(), <U625 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU5 = <<A as BitAnd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitAndU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitXorU5 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitXorU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1

    #[allow(non_camel_case_types)]
    type U5ShlU5 = <<A as Shl<B>>::Output as Same<U160>>::Output;

    assert_eq!(<U5ShlU5 as Unsigned>::to_u64(), <U160 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU5 = <<A as Add<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5AddU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU5 = <<A as Min<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MinU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5SubU5 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5SubU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU5 = <<A as Mul<B>>::Output as Same<U25>>::Output;
```

```
    assert_eq!(<U5MulU5 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU5 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U5PartialDivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type U5PowU5 = <<A as Pow<B>>::Output as Same<U3125>>::Output;

    assert_eq!(<U5PowU5 as Unsigned>::to_u64(), <U3125 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN5 = <<A as Add<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5AddN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5SubN5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5SubN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulN5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5MulN5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN5 = <<A as Max<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MaxN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N5PartialDivN5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN4 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5AddN4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5SubN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulN4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N5MulN4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N5DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN3 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5AddN3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5SubN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N3() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N5MulN3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN2 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5AddN2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5SubN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulN2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N5MulN2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN1 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5AddN1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5SubN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MulN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N1() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5DivN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<N5PartialDivN1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Add_0 = <<A as Add<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5Add_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Sub_0 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Sub_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Min_0 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Min_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N5Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5AddP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N5SubP1 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5SubP1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MulP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5DivP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<N5PartialDivP1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PowP1 = <<A as Pow<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5PowP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P1() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5AddP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP2 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5SubP2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulP2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5MulP2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP2 = <<A as Min<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5MinP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5AddP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP3 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5SubP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MulP3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N5MulP3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N5MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type N5PowP3 = <<A as Pow<B>>::Output as Same<N125>>::Output;

    assert_eq!(<N5PowP3 as Integer>::to_i64(), <N125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5AddP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP4 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5SubP4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulP4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N5MulP4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N5MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type N5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<N5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P5() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5AddP5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5AddP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP5 = <<A as Sub<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5SubP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulP5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<N5MulP5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N5PartialDivP5 as Integer>::to_i64(), <N1 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N5PowP5 = <<A as Pow<B>>::Output as Same<N3125>>::Output;

    assert_eq!(<N5PowP5 as Integer>::to_i64(), <N3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN5 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4AddN5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN5 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4SubN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N4MulN5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN5 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemN5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN4 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4AddN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4SubN4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4SubN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N4() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4MulN4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N4DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N4PartialDivN4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN3 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4AddN3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4SubN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N4MulN3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN2 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4AddN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubN2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4SubN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N4MulN2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N2() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N4PartialDivN2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN1 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4AddN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4SubN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MulN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4DivN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<N4PartialDivN1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Add_0 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Add_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Sub_0 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Sub_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Min_0 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Min_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp__0() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N4Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4AddP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP1 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4SubP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MulP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP1 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4MinP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4DivP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<N4PartialDivP1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PowP1 = <<A as Pow<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4PowP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddP2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4AddP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4SubP2 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4SubP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4MulP2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N4PartialDivP2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P2() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4AddP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP3 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4SubP3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N4MulP3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP3 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4MinP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N4MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N64 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>
    #[allow(non_camel_case_types)]
    type N4PowP3 = <<A as Pow<B>>::Output as Same<N64>>::Output;
    assert_eq!(<N4PowP3 as Integer>::to_i64(), <N64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    #[allow(non_camel_case_types)]
    type N4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
    #[allow(non_camel_case_types)]
    type N4AddP4 = <<A as Add<B>>::Output as Same<_0>>::Output;
    assert_eq!(<N4AddP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
    #[allow(non_camel_case_types)]
    type N4SubP4 = <<A as Sub<B>>::Output as Same<N8>>::Output;
    assert_eq!(<N4SubP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type N4MulP4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<N4MulP4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N4PartialDivP4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type N4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<N4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP5 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4AddP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP5 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4SubP5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N4MulP5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP5 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<N4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemP5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1024 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N4PowP5 = <<A as Pow<B>>::Output as Same<N1024>>::Output;

    assert_eq!(<N4PowP5 as Integer>::to_i64(), <N1024 as Integer>::to_i64()
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN5 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3AddN5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN5 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3SubN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N3MulN5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN5 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN4 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3AddN4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3SubN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N3MulN4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN4 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN4 = <<A as Rem<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3RemN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN3 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3AddN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3SubN3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3SubN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3MulN3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N3() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N3PartialDivN3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN2 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3AddN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3SubN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N3MulN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3AddN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3SubN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MulN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N1() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3DivN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<N3PartialDivN1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Add_0 = <<A as Add<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3Add_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Sub_0 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Sub_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Min_0 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Min_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N3Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3AddP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N3SubP1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3SubP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MulP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3DivP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<N3PartialDivP1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP1 = <<A as Pow<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3PowP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P1() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3AddP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP2 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3SubP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3MulP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP2 = <<A as Min<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3MinP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3AddP3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3AddP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP3 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3SubP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MulP3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N3MulP3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N3PartialDivP3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N27 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP3 = <<A as Pow<B>>::Output as Same<N27>>::Output;

    assert_eq!(<N3PowP3 as Integer>::to_i64(), <N27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3AddP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP4 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3SubP4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N3MulP4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP4 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<N3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<N3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP5 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3AddP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP5 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3SubP5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N3MulP5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinP5 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N243 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type N3PowP5 = <<A as Pow<B>>::Output as Same<N243>>::Output;

    assert_eq!(<N3PowP5 as Integer>::to_i64(), <N243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN5 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2AddN5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN5 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2SubN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N2MulN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN5 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN4 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2AddN4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubN4 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2SubN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N4() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N2MulN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN4 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN3 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2AddN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2SubN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N2MulN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN3 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN2 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2AddN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2SubN2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2SubN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MulN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N2() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N2PartialDivN2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN1 = <<A as Add<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N2AddN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2SubN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MulN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivN1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2DivN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N2PartialDivN1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Add_0 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Add_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Sub_0 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Sub_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Min_0 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Min_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N2Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2AddP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2SubP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MulP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N2MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivP1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2DivP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N2PartialDivP1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP1 = <<A as Pow<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2PowP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2AddP2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2AddP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP2 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2SubP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MulP2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MulP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N2PartialDivP2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2AddP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP3 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2SubP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2MulP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP3 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<N2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP3 = <<A as Pow<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2PowP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddP4 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2AddP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP4 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2SubP4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2MulP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MinP4 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP5 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2AddP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP5 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2SubP5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N2MulP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP5 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N32 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type N2PowP5 = <<A as Pow<B>>::Output as Same<N32>>::Output;

    assert_eq!(<N2PowP5 as Integer>::to_i64(), <N32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN5 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1AddN5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N5() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN5 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1SubN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MulN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN5 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Add_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN4 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1AddN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN4 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1SubN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MulN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN4 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN3 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1AddN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1SubN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MulN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N3() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN3 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1AddN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1SubN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Mul_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MulN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN2 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1AddN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1SubN1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1SubN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MulN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N1() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivN1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1DivN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N1PartialDivN1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1PowN1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Add_0 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Add_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Sub_0 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Sub_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Min__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Min_0 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Min_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N1Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1AddP1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1AddP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1SubP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MulP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivP1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1DivP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_P1() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N1PartialDivP1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1AddP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP2 = <<A as Sub<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N1SubP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MulP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP2 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Div_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N1AddP3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1AddP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP3 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1SubP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MulP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP3 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P3() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP4 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1AddP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP4 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1SubP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MulP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP4 = <<A as Min<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N1MinP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Pow_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP5 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1AddP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP5 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1SubP5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1MulP5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MulP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP5 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN5 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0AddN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN5 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0SubN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN5 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<_0GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N4() {
```

```
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN4 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0AddN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN4 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0SubN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0MaxN4 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN4 as Integer>::to_i64(), <_0 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN3 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0AddN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN3 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0SubN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN3 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0AddN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0SubN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N2() {
```

```
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN2 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0AddN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Sub_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0SubN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN1 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN1 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Add_0 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Add_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Sub_0 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Sub_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max__0() {
```

```
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Gcd_0 = <<A as Gcd<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Gcd_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow__0() {
    type A = Z0;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp__0() {
    type A = Z0;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type _0Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<_0Cmp_0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP1 = <<A as Add<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<_0AddP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0SubP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP1 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Gcd_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP1 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0AddP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0SubP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP2 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P2() {
```

```
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP2 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP3 = <<A as Add<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<_0AddP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP3 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0SubP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP3 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Gcd_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP3 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP4 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0AddP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP4 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0SubP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP4 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P4() {
```

```
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP4 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP5 = <<A as Add<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<_0AddP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP5 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0SubP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP5 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Gcd_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP5 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN5 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1AddN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN5 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1SubN5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MulN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN5 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N5() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN4 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1AddN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN4 = <<A as Sub<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P1SubN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MulN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN4 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Div_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P1AddN3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1AddN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN3 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1SubN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MulN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN3 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N3() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1AddN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1SubN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MulN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<P1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN2 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Pow_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1AddN1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1AddN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1SubN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1MulN1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MulN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivN1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1DivN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P1PartialDivN1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Add_0 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Add_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Sub_0 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Sub_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Max_0 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1Max_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P1Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1AddP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P1() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1SubP1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1SubP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MulP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivP1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1DivP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P1PartialDivP1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1AddP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1SubP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MulP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP2 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP3 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1AddP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1SubP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P3() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MulP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP3 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type P1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP4 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1AddP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Sub_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP4 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1SubP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MulP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP4 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP4 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP5 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1AddP5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP5 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1SubP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MulP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP5 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P5() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1PowP5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN5 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2AddN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN5 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2SubN5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P2MulN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN5 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddN4 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2AddN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN4 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2SubN4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P2MulN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN4 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N4() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2AddN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN3 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2SubN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN3 = <<A as Mul<B>>::Output as Same<N6>>::Output;
```

```
    assert_eq!(<P2MulN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN3 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2AddN2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2AddN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN2 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2SubN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2MulN2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MulN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P2PartialDivN2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2AddN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2SubN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MulN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivN1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2DivN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P2PartialDivN1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add__0() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Add_0 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Add_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Sub_0 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Sub_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2Max_0 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Max_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P2Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2AddP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2SubP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MulP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivP1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2DivP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P2PartialDivP1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP1 = <<A as Pow<B>>::Output as Same<P2>>::Output;
```

```
    assert_eq!(<P2PowP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP2 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2AddP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2SubP2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2SubP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MulP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P2() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type P2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P2PartialDivP2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP3 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2AddP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2SubP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2MulP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP3 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP3 = <<A as Pow<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2PowP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP3 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP4 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2AddP4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubP4 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2SubP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2MulP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP4 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P4() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type P2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP5 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2AddP5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP5 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2SubP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P2MulP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP5 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P32 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP5 = <<A as Pow<B>>::Output as Same<P32>>::Output;

    assert_eq!(<P2PowP5 as Integer>::to_i64(), <P32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN5 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3AddN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN5 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3SubN5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P3MulN5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN5 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N5() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3AddN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN4 = <<A as Sub<B>>::Output as Same<P7>>::Output;
```

```
    assert_eq!(<P3SubN4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P3MulN4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN4 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3AddN3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3AddN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P3SubN3 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3SubN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<P3MulN3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P3PartialDivN3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3AddN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN2 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3SubN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P3MulN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN2 = <<A as Max<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<P3MaxN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N1() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3AddN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3SubN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MulN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3MaxN1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3DivN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<P3PartialDivN1 as Integer>::to_i64(), <N3 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Add_0 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Add_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Sub_0 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Sub_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Max_0 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Max_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P3Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3AddP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3SubP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MulP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P1() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3DivP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<P3PartialDivP1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP1 = <<A as Pow<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3PowP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP2 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3AddP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3SubP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3MulP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP3 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3AddP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3SubP3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3SubP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3MulP3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P3() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P3PartialDivP3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P27 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>
```

```
    #[allow(non_camel_case_types)]
    type P3PowP3 = <<A as Pow<B>>::Output as Same<P27>>::Output;

    assert_eq!(<P3PowP3 as Integer>::to_i64(), <P27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP4 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3AddP4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3SubP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P3MulP4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP4 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<P3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP5 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3AddP5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP5 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3SubP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P3MulP5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP5 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P5() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P243 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type P3PowP5 = <<A as Pow<B>>::Output as Same<P243>>::Output;

    assert_eq!(<P3PowP5 as Integer>::to_i64(), <P243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN5 = <<A as Add<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P4AddN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN5 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4SubN5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P4MulN5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN5 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemN5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type P4AddN4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4AddN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN4 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4SubN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<P4MulN4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P4PartialDivN4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N4() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4AddN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN3 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4SubN3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P4MulN3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<P4MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddN2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4AddN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN2 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4SubN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P4MulN2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P4PartialDivN2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4AddN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN1 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4SubN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MulN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN1 = <<A as Div<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<P4DivN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<P4PartialDivN1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Add_0 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Add_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub__0() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Sub_0 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Sub_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Max_0 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Max_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P4Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP1 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4AddP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4SubP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MulP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4DivP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<P4PartialDivP1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PowP1 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4PowP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP1 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP2 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4AddP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubP2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4SubP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4MulP2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P2() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P4PartialDivP2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP3 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4AddP3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4SubP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P4MulP3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P64 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP3 = <<A as Pow<B>>::Output as Same<P64>>::Output;

    assert_eq!(<P4PowP3 as Integer>::to_i64(), <P64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP4 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4AddP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4SubP4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4SubP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4MulP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P4() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P4PartialDivP4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1
```

```
    #[allow(non_camel_case_types)]
    type P4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<P4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP5 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4AddP5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP5 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4SubP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P4MulP5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP5 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemP5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1024 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P4PowP5 = <<A as Pow<B>>::Output as Same<P1024>>::Output;

    assert_eq!(<P4PowP5 as Integer>::to_i64(), <P1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5AddN5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5AddN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN5 = <<A as Sub<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5SubN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulN5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<P5MulN5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N5() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P5PartialDivN5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN4 = <<A as Add<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P5AddN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN4 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5SubN4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulN4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P5MulN4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P5MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P5AddN3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5AddN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN3 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5SubN3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P5MulN3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P5MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5AddN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN2 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5SubN2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulN2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P5MulN2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN2 = <<A as Max<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P5MaxN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N1() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5AddN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN1 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5SubN1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MulN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5MaxN1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5DivN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<P5PartialDivN1 as Integer>::to_i64(), <N5 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Add_0 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Add_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Sub_0 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Sub_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Max_0 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Max_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P5Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP1 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5AddP1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5SubP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MulP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P1() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5DivP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<P5PartialDivP1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PowP1 = <<A as Pow<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5PowP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP2 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5AddP2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5SubP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulP2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5MulP2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP3 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5AddP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5SubP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P5MulP3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P3() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1
```

```
    #[allow(non_camel_case_types)]
    type P5PowP3 = <<A as Pow<B>>::Output as Same<P125>>::Output;

    assert_eq!(<P5PowP3 as Integer>::to_i64(), <P125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP4 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5AddP4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5SubP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulP4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P5MulP4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type P5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<P5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP5 = <<A as Add<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5AddP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5SubP5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5SubP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulP5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5MulP5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP5 = <<A as Min<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MinP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P5() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P5PartialDivP5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P5PowP5 = <<A as Pow<B>>::Output as Same<P3125>>::Output;

    assert_eq!(<P5PowP5 as Integer>::to_i64(), <P3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN5 = <<A as Neg>::Output as Same<P5>>::Output;
    assert_eq!(<NegN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN4 = <<A as Neg>::Output as Same<P4>>::Output;
    assert_eq!(<NegN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN3 = <<A as Neg>::Output as Same<P3>>::Output;
    assert_eq!(<NegN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN2 = <<A as Neg>::Output as Same<P2>>::Output;
    assert_eq!(<NegN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Neg() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegN1 = <<A as Neg>::Output as Same<P1>>::Output;
    assert_eq!(<NegN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Abs() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Neg() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Neg_0 = <<A as Neg>::Output as Same<_0>>::Output;
    assert_eq!(<Neg_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Abs() {
```

```
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Abs_0 = <<A as Abs>::Output as Same<_0>>::Output;
    assert_eq!(<Abs_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Neg() {
    type A = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegP1 = <<A as Neg>::Output as Same<N1>>::Output;
    assert_eq!(<NegP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Abs() {
    type A = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP2 = <<A as Neg>::Output as Same<N2>>::Output;
    assert_eq!(<NegP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type NegP3 = <<A as Neg>::Output as Same<N3>>::Output;
    assert_eq!(<NegP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP4 = <<A as Neg>::Output as Same<N4>>::Output;
    assert_eq!(<NegP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP5 = <<A as Neg>::Output as Same<N5>>::Output;
    assert_eq!(<NegP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type AbsP5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

File: ./target/x86_64-pc-windows-gnu/debug/build/typenum-8a82b59

```
/**
Convenient type operations.

Any types representing values must be able to be expressed as `ident`s. Tha
in scope.

For example, `P5` is okay, but `typenum::P5` is not.

You may combine operators arbitrarily, although doing so excessively may re
recursion limit.

# Example
```rust
#![recursion_limit="128"]
#[macro_use] extern crate typenum;
use typenum::consts::*;

fn main() {
    assert_type!(
        op!(min((P1 - P2) * (N3 + N7), P5 * (P3 + P4)) == P10)
    );
}
```
Operators are evaluated based on the operator precedence outlined
[here](https://doc.rust-lang.org/reference.html#operator-precedence).

The full list of supported operators and functions is as follows:

`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, `==`, `!=`, `<=`, `>=`,

They all expand to type aliases defined in the `operator_aliases` module. H
including examples:

Operator `*`. Expands to `Prod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 * P3), P6);
# }
```
```

Operator `/`. Expands to `Quot`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 / P2), P3);
# }
```

Operator `%`. Expands to `Mod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 % P3), P2);
# }
```

Operator `+`. Expands to `Sum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 + P3), P5);
# }
```

Operator `-`. Expands to `Diff`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 - P3), N1);
# }
```

Operator `<<`. Expands to `Shleft`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
```

```
assert_type_eq!(op!(U1 << U5), U32);
# }
```

Operator `>>`. Expands to `Shright`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U32 >> U5), U1);
# }
```

Operator `&`. Expands to `And`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 & U3), U1);
# }
```

Operator `^`. Expands to `Xor`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 ^ U3), U6);
# }
```

Operator `|`. Expands to `Or`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 | U3), U7);
# }
```

Operator `==`. Expands to `Eq`.

```rust
```

```
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 == P3 + P2), True);
# }
```

Operator `!=`. Expands to `NotEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 != P3 + P2), False);
# }
```

Operator `<=`. Expands to `LeEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 <= P3 + P2), False);
# }
```

Operator `>=`. Expands to `GrEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 >= P3 + P2), True);
# }
```

Operator `<`. Expands to `Le`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P4 < P3 + P2), True);
# }
```

Operator `>`. Expands to `Gr`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 < P3 + P2), False);
# }
```

Operator `cmp`. Expands to `Compare`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cmp(P2, P3)), Less);
# }
```

Operator `sqr`. Expands to `Square`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqr(P2)), P4);
# }
```

Operator `sqrt`. Expands to `Sqrt`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqrt(U9)), U3);
# }
```

Operator `abs`. Expands to `AbsVal`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(abs(N2)), P2);
# }
```

```

---
Operator `cube`. Expands to `Cube`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cube(P2)), P8);
# }
```

Operator `pow`. Expands to `Exp`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(pow(P2, P3)), P8);
# }
```

Operator `min`. Expands to `Minimum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(min(P2, P3)), P2);
# }
```

Operator `max`. Expands to `Maximum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(max(P2, P3)), P3);
# }
```

Operator `log2`. Expands to `Log2`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
```
```

```
# fn main() {
assert_type_eq!(op!(log2(U9)), U3);
# }
```

Operator `gcd`. Expands to `Gcf`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(gcd(U9, U21)), U3);
# }
```

*/
#[macro_export(local_inner_macros)]
macro_rules! op {
    ($($tail:tt)*) => (__op_internal__!($($tail)*));
}

    #[doc(hidden)]
    #[macro_export(local_inner_macros)]
    macro_rules! __op_internal__ {

(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cmp $($tail:tt)
    __op_internal__!(@stack[Compare, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqr $($tail:tt)
    __op_internal__!(@stack[Square, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqrt $($tail:tt
    __op_internal__!(@stack[Sqrt, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: abs $($tail:tt)
    __op_internal__!(@stack[AbsVal, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cube $($tail:tt
    __op_internal__!(@stack[Cube, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: pow $($tail:tt)
    __op_internal__!(@stack[Exp, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: min $($tail:tt)
    __op_internal__!(@stack[Minimum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: max $($tail:tt)
    __op_internal__!(@stack[Maximum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: log2 $($tail:tt
    __op_internal__!(@stack[Log2, $($stack,)*] @queue[$($queue,)*] @tail: $

```
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: gcd $($tail:tt)
    __op_internal__!(@stack[Gcf, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: , $($ta
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: *
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: *
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: *
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:tt)*)
    __op_internal__!(@stack[Prod, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: /
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: /
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: /
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:tt)*)
    __op_internal__!(@stack[Quot, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: %
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: %
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: %
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:tt)*)
    __op_internal__!(@stack[Mod, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: +
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: +
);
```

```
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: +
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: +
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: +
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:tt)*)
    __op_internal__!(@stack[Sum, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: -
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: -
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: -
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: -
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: -
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:tt)*)
    __op_internal__!(@stack[Diff, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <<
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <<
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail:tt)*
```

```
    __op_internal__!(@stack[Shleft, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >>
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >>
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail:tt)*
    __op_internal__!(@stack[Shright, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: &
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: &
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: &
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: &
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: &
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: &
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:tt)*)
    __op_internal__!(@stack[And, $($stack,)*] @queue[$($queue,)*] @tail: $(
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: ^
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: ^
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ^
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ^
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: ^
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ^
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ^
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:tt)*)
    __op_internal__!(@stack[Xor, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: |
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: |
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: |
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: |
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: |
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
```

```
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: |
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: |
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: | $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:tt)*)
    __op_internal__!(@stack[Or, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: =
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: =
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ==
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ==
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: =
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ==
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ==
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: ==
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: ==
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: =
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: =
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: ==
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: ==
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:tt)*
    __op_internal__!(@stack[Eq, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: !
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: !
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: !=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: !=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: !
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: !=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: !=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: !=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: !=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: !
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: !
```

```
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: !=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: !=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:tt)*
    __op_internal__!(@stack[NotEq, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: <=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: <=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
```

```
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: <=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: <=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:tt)*
    __op_internal__!(@stack[LeEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: >=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: >=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: >=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: >=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:tt)*
    __op_internal__!(@stack[GrEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: < $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: < $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: < $
```

```
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: < $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:tt)*)
    __op_internal__!(@stack[Le, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: > $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: > $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: > $
);
```

```
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: > $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:tt)*)
    __op_internal__!(@stack[Gr, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ( $($stuff:tt)*
 => (
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*]
                     @tail: $($stuff)* RParen $($tail)*)
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: RParen
    __op_internal__!(@rp3 @stack[$($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
 => (
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@rp3 @stack[Compare, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Compare, $($queue,)*] @tail
);
(@rp3 @stack[Square, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[Square, $($queue,)*] @tail:
);
(@rp3 @stack[Sqrt, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Sqrt, $($queue,)*] @tail: $
);
(@rp3 @stack[AbsVal, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[AbsVal, $($queue,)*] @tail:
);
(@rp3 @stack[Cube, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Cube, $($queue,)*] @tail: $
);
(@rp3 @stack[Exp, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Exp, $($queue,)*] @tail: $(
);
(@rp3 @stack[Minimum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Minimum, $($queue,)*] @tail
);
(@rp3 @stack[Maximum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Maximum, $($queue,)*] @tail
);
(@rp3 @stack[Log2, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Log2, $($queue,)*] @tail: $
);
(@rp3 @stack[Gcf, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Gcf, $($queue,)*] @tail: $(
);
(@rp3 @stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($tail:tt
    __op_internal__!(@stack[$($stack,)*] @queue[$($queue,)*] @tail: $($tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $num:ident $($t
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[$num, $($queue,)*] @tail: $
);
(@stack[] @queue[$($queue:ident,)*] @tail: ) => (
    __op_internal__!(@reverse[] @input: $($queue,)*)
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@reverse[$($revved:ident,)*] @input: $head:ident, $($tail:ident,)* ) => (
    __op_internal__!(@reverse[$head, $($revved,)*] @input: $($tail,)*)
);
(@reverse[$($revved:ident,)*] @input: ) => (
    __op_internal__!(@eval @stack[] @input[$($revved,)*])
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Prod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Prod<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Quot, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Quot<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Mod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Mod<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Sum, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Sum<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Diff, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Diff<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shleft, $($tail:ident,)*
    __op_internal__!(@eval @stack[$crate::Shleft<$b, $a>, $($stack,)*] @inp
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shright, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Shright<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[And, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::And<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Xor, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Xor<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Or, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Or<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Eq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Eq<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[NotEq, $($tail:ident,)*]
    __op_internal__!(@eval @stack[$crate::NotEq<$b, $a>, $($stack,)*] @inpu
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[LeEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::LeEq<$b, $a>, $($stack,)*] @input
```

```
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[GrEq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::GrEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Le, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Le<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gr, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Gr<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Compare, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Compare<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Exp, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Exp<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Minimum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Minimum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Maximum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Maximum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gcf, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Gcf<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Square, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Square<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Sqrt, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Sqrt<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[AbsVal, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::AbsVal<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Cube, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Cube<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Log2, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Log2<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$($stack:ty,)*] @input[$head:ident, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$head, $($stack,)*] @input[$($tail,)*])
);
(@eval @stack[$stack:ty,] @input[]) => (
    $stack
);
($($tail:tt)* ) => (
    __op_internal__!(@stack[] @queue[] @tail: $($tail)*)
);
}
```

File: ./target/package/catalysh-0.0.2/target/debug/build/typenum-056

```
/**
Type aliases for many constants.

This file is generated by typenum's build script.

For unsigned integers, the format is `U` followed by the number. We define

- Numbers 0 through 1024
- Powers of 2 below `u64::MAX`
- Powers of 10 below `u64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm};

# #[allow(dead_code)]
type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

For positive signed integers, the format is `P` followed by the number and
signed integers it is `N` followed by the number. For the signed integer ze
`Z0`. We define aliases for

- Numbers -1024 through 1024
- Powers of 2 between `i64::MIN` and `i64::MAX`
- Powers of 10 between `i64::MIN` and `i64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm, PInt, NInt};

# #[allow(dead_code)]
type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
# #[allow(dead_code)]
type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```


Example
```rust
# #[allow(unused_imports)]
use typenum::{U0, U1, U2, U3, U4, U5, U6};
# #[allow(unused_imports)]
use typenum::{N3, N2, N1, Z0, P1, P2, P3};
# #[allow(unused_imports)]
use typenum::{U774, N17, N10000, P1024, P4096};
```
```

```
We also define the aliases `False` and `True` for `B0` and `B1`, respective
*/
#[allow(missing_docs)]
pub mod consts {
    use crate::uint::{UInt, UTerm};
    use crate::int::{PInt, NInt};

    pub use crate::bit::{B0, B1};
    pub use crate::int::Z0;

    pub type True = B1;
    pub type False = B0;
    pub type U0 = UTerm;
    pub type U1 = UInt<UTerm, B1>;
    pub type P1 = PInt<U1>; pub type N1 = NInt<U1>;
    pub type U2 = UInt<UInt<UTerm, B1>, B0>;
    pub type P2 = PInt<U2>; pub type N2 = NInt<U2>;
    pub type U3 = UInt<UInt<UTerm, B1>, B1>;
    pub type P3 = PInt<U3>; pub type N3 = NInt<U3>;
    pub type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    pub type P4 = PInt<U4>; pub type N4 = NInt<U4>;
    pub type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    pub type P5 = PInt<U5>; pub type N5 = NInt<U5>;
    pub type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
    pub type P6 = PInt<U6>; pub type N6 = NInt<U6>;
    pub type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
    pub type P7 = PInt<U7>; pub type N7 = NInt<U7>;
    pub type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
    pub type P8 = PInt<U8>; pub type N8 = NInt<U8>;
    pub type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;
    pub type P9 = PInt<U9>; pub type N9 = NInt<U9>;
    pub type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;
    pub type P10 = PInt<U10>; pub type N10 = NInt<U10>;
    pub type U11 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>;
    pub type P11 = PInt<U11>; pub type N11 = NInt<U11>;
    pub type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;
    pub type P12 = PInt<U12>; pub type N12 = NInt<U12>;
    pub type U13 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>;
    pub type P13 = PInt<U13>; pub type N13 = NInt<U13>;
    pub type U14 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>;
    pub type P14 = PInt<U14>; pub type N14 = NInt<U14>;
    pub type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;
    pub type P15 = PInt<U15>; pub type N15 = NInt<U15>;
    pub type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
    pub type P16 = PInt<U16>; pub type N16 = NInt<U16>;
    pub type U17 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>;
    pub type P17 = PInt<U17>; pub type N17 = NInt<U17>;
    pub type U18 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B0>;
    pub type P18 = PInt<U18>; pub type N18 = NInt<U18>;
    pub type U19 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B1>;
    pub type P19 = PInt<U19>; pub type N19 = NInt<U19>;
    pub type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;
```

```
pub type P20 = PInt<U20>; pub type N20 = NInt<U20>;
pub type U21 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B1>;
pub type P21 = PInt<U21>; pub type N21 = NInt<U21>;
pub type U22 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B0>;
pub type P22 = PInt<U22>; pub type N22 = NInt<U22>;
pub type U23 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B1>;
pub type P23 = PInt<U23>; pub type N23 = NInt<U23>;
pub type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;
pub type P24 = PInt<U24>; pub type N24 = NInt<U24>;
pub type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
pub type P25 = PInt<U25>; pub type N25 = NInt<U25>;
pub type U26 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B0>;
pub type P26 = PInt<U26>; pub type N26 = NInt<U26>;
pub type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;
pub type P27 = PInt<U27>; pub type N27 = NInt<U27>;
pub type U28 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B0>;
pub type P28 = PInt<U28>; pub type N28 = NInt<U28>;
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
pub type U35 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P35 = PInt<U35>; pub type N35 = NInt<U35>;
pub type U36 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P36 = PInt<U36>; pub type N36 = NInt<U36>;
pub type U37 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P37 = PInt<U37>; pub type N37 = NInt<U37>;
pub type U38 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P38 = PInt<U38>; pub type N38 = NInt<U38>;
pub type U39 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P39 = PInt<U39>; pub type N39 = NInt<U39>;
pub type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P40 = PInt<U40>; pub type N40 = NInt<U40>;
pub type U41 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P41 = PInt<U41>; pub type N41 = NInt<U41>;
pub type U42 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P42 = PInt<U42>; pub type N42 = NInt<U42>;
pub type U43 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P43 = PInt<U43>; pub type N43 = NInt<U43>;
pub type U44 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P44 = PInt<U44>; pub type N44 = NInt<U44>;
pub type U45 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P45 = PInt<U45>; pub type N45 = NInt<U45>;
pub type U46 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
```

```
pub type P46 = PInt<U46>; pub type N46 = NInt<U46>;
pub type U47 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P47 = PInt<U47>; pub type N47 = NInt<U47>;
pub type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P48 = PInt<U48>; pub type N48 = NInt<U48>;
pub type U49 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P49 = PInt<U49>; pub type N49 = NInt<U49>;
pub type U50 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P50 = PInt<U50>; pub type N50 = NInt<U50>;
pub type U51 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P51 = PInt<U51>; pub type N51 = NInt<U51>;
pub type U52 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P52 = PInt<U52>; pub type N52 = NInt<U52>;
pub type U53 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P53 = PInt<U53>; pub type N53 = NInt<U53>;
pub type U54 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P54 = PInt<U54>; pub type N54 = NInt<U54>;
pub type U55 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P55 = PInt<U55>; pub type N55 = NInt<U55>;
pub type U56 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P56 = PInt<U56>; pub type N56 = NInt<U56>;
pub type U57 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P57 = PInt<U57>; pub type N57 = NInt<U57>;
pub type U58 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P58 = PInt<U58>; pub type N58 = NInt<U58>;
pub type U59 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P59 = PInt<U59>; pub type N59 = NInt<U59>;
pub type U60 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P60 = PInt<U60>; pub type N60 = NInt<U60>;
pub type U61 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P61 = PInt<U61>; pub type N61 = NInt<U61>;
pub type U62 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P62 = PInt<U62>; pub type N62 = NInt<U62>;
pub type U63 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P63 = PInt<U63>; pub type N63 = NInt<U63>;
pub type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P64 = PInt<U64>; pub type N64 = NInt<U64>;
pub type U65 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P65 = PInt<U65>; pub type N65 = NInt<U65>;
pub type U66 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P66 = PInt<U66>; pub type N66 = NInt<U66>;
pub type U67 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P67 = PInt<U67>; pub type N67 = NInt<U67>;
pub type U68 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P68 = PInt<U68>; pub type N68 = NInt<U68>;
pub type U69 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P69 = PInt<U69>; pub type N69 = NInt<U69>;
pub type U70 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P70 = PInt<U70>; pub type N70 = NInt<U70>;
pub type U71 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P71 = PInt<U71>; pub type N71 = NInt<U71>;
pub type U72 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
```

```
pub type P72 = PInt<U72>; pub type N72 = NInt<U72>;
pub type U73 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P73 = PInt<U73>; pub type N73 = NInt<U73>;
pub type U74 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P74 = PInt<U74>; pub type N74 = NInt<U74>;
pub type U75 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P75 = PInt<U75>; pub type N75 = NInt<U75>;
pub type U76 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P76 = PInt<U76>; pub type N76 = NInt<U76>;
pub type U77 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P77 = PInt<U77>; pub type N77 = NInt<U77>;
pub type U78 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P78 = PInt<U78>; pub type N78 = NInt<U78>;
pub type U79 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P79 = PInt<U79>; pub type N79 = NInt<U79>;
pub type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P80 = PInt<U80>; pub type N80 = NInt<U80>;
pub type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P81 = PInt<U81>; pub type N81 = NInt<U81>;
pub type U82 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P82 = PInt<U82>; pub type N82 = NInt<U82>;
pub type U83 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P83 = PInt<U83>; pub type N83 = NInt<U83>;
pub type U84 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P84 = PInt<U84>; pub type N84 = NInt<U84>;
pub type U85 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P85 = PInt<U85>; pub type N85 = NInt<U85>;
pub type U86 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P86 = PInt<U86>; pub type N86 = NInt<U86>;
pub type U87 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P87 = PInt<U87>; pub type N87 = NInt<U87>;
pub type U88 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P88 = PInt<U88>; pub type N88 = NInt<U88>;
pub type U89 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P89 = PInt<U89>; pub type N89 = NInt<U89>;
pub type U90 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P90 = PInt<U90>; pub type N90 = NInt<U90>;
pub type U91 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P91 = PInt<U91>; pub type N91 = NInt<U91>;
pub type U92 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P92 = PInt<U92>; pub type N92 = NInt<U92>;
pub type U93 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P93 = PInt<U93>; pub type N93 = NInt<U93>;
pub type U94 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P94 = PInt<U94>; pub type N94 = NInt<U94>;
pub type U95 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P95 = PInt<U95>; pub type N95 = NInt<U95>;
pub type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P96 = PInt<U96>; pub type N96 = NInt<U96>;
pub type U97 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P97 = PInt<U97>; pub type N97 = NInt<U97>;
pub type U98 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
```

```
pub type P98 = PInt<U98>; pub type N98 = NInt<U98>;
pub type U99 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P99 = PInt<U99>; pub type N99 = NInt<U99>;
pub type U100 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P100 = PInt<U100>; pub type N100 = NInt<U100>;
pub type U101 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P101 = PInt<U101>; pub type N101 = NInt<U101>;
pub type U102 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P102 = PInt<U102>; pub type N102 = NInt<U102>;
pub type U103 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P103 = PInt<U103>; pub type N103 = NInt<U103>;
pub type U104 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P104 = PInt<U104>; pub type N104 = NInt<U104>;
pub type U105 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P105 = PInt<U105>; pub type N105 = NInt<U105>;
pub type U106 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P106 = PInt<U106>; pub type N106 = NInt<U106>;
pub type U107 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P107 = PInt<U107>; pub type N107 = NInt<U107>;
pub type U108 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P108 = PInt<U108>; pub type N108 = NInt<U108>;
pub type U109 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P109 = PInt<U109>; pub type N109 = NInt<U109>;
pub type U110 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P110 = PInt<U110>; pub type N110 = NInt<U110>;
pub type U111 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P111 = PInt<U111>; pub type N111 = NInt<U111>;
pub type U112 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P112 = PInt<U112>; pub type N112 = NInt<U112>;
pub type U113 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P113 = PInt<U113>; pub type N113 = NInt<U113>;
pub type U114 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P114 = PInt<U114>; pub type N114 = NInt<U114>;
pub type U115 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P115 = PInt<U115>; pub type N115 = NInt<U115>;
pub type U116 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P116 = PInt<U116>; pub type N116 = NInt<U116>;
pub type U117 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P117 = PInt<U117>; pub type N117 = NInt<U117>;
pub type U118 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P118 = PInt<U118>; pub type N118 = NInt<U118>;
pub type U119 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P119 = PInt<U119>; pub type N119 = NInt<U119>;
pub type U120 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P120 = PInt<U120>; pub type N120 = NInt<U120>;
pub type U121 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P121 = PInt<U121>; pub type N121 = NInt<U121>;
pub type U122 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P122 = PInt<U122>; pub type N122 = NInt<U122>;
pub type U123 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P123 = PInt<U123>; pub type N123 = NInt<U123>;
pub type U124 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
```

```
pub type P124 = PInt<U124>; pub type N124 = NInt<U124>;
pub type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P125 = PInt<U125>; pub type N125 = NInt<U125>;
pub type U126 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P126 = PInt<U126>; pub type N126 = NInt<U126>;
pub type U127 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P127 = PInt<U127>; pub type N127 = NInt<U127>;
pub type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P128 = PInt<U128>; pub type N128 = NInt<U128>;
pub type U129 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P129 = PInt<U129>; pub type N129 = NInt<U129>;
pub type U130 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P130 = PInt<U130>; pub type N130 = NInt<U130>;
pub type U131 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P131 = PInt<U131>; pub type N131 = NInt<U131>;
pub type U132 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P132 = PInt<U132>; pub type N132 = NInt<U132>;
pub type U133 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P133 = PInt<U133>; pub type N133 = NInt<U133>;
pub type U134 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P134 = PInt<U134>; pub type N134 = NInt<U134>;
pub type U135 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P135 = PInt<U135>; pub type N135 = NInt<U135>;
pub type U136 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P136 = PInt<U136>; pub type N136 = NInt<U136>;
pub type U137 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P137 = PInt<U137>; pub type N137 = NInt<U137>;
pub type U138 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P138 = PInt<U138>; pub type N138 = NInt<U138>;
pub type U139 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P139 = PInt<U139>; pub type N139 = NInt<U139>;
pub type U140 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P140 = PInt<U140>; pub type N140 = NInt<U140>;
pub type U141 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P141 = PInt<U141>; pub type N141 = NInt<U141>;
pub type U142 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P142 = PInt<U142>; pub type N142 = NInt<U142>;
pub type U143 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P143 = PInt<U143>; pub type N143 = NInt<U143>;
pub type U144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P144 = PInt<U144>; pub type N144 = NInt<U144>;
pub type U145 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P145 = PInt<U145>; pub type N145 = NInt<U145>;
pub type U146 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P146 = PInt<U146>; pub type N146 = NInt<U146>;
pub type U147 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P147 = PInt<U147>; pub type N147 = NInt<U147>;
pub type U148 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P148 = PInt<U148>; pub type N148 = NInt<U148>;
pub type U149 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P149 = PInt<U149>; pub type N149 = NInt<U149>;
pub type U150 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
```

```
pub type P150 = PInt<U150>; pub type N150 = NInt<U150>;
pub type U151 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P151 = PInt<U151>; pub type N151 = NInt<U151>;
pub type U152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P152 = PInt<U152>; pub type N152 = NInt<U152>;
pub type U153 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P153 = PInt<U153>; pub type N153 = NInt<U153>;
pub type U154 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P154 = PInt<U154>; pub type N154 = NInt<U154>;
pub type U155 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P155 = PInt<U155>; pub type N155 = NInt<U155>;
pub type U156 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P156 = PInt<U156>; pub type N156 = NInt<U156>;
pub type U157 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P157 = PInt<U157>; pub type N157 = NInt<U157>;
pub type U158 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P158 = PInt<U158>; pub type N158 = NInt<U158>;
pub type U159 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P159 = PInt<U159>; pub type N159 = NInt<U159>;
pub type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P160 = PInt<U160>; pub type N160 = NInt<U160>;
pub type U161 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P161 = PInt<U161>; pub type N161 = NInt<U161>;
pub type U162 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P162 = PInt<U162>; pub type N162 = NInt<U162>;
pub type U163 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P163 = PInt<U163>; pub type N163 = NInt<U163>;
pub type U164 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P164 = PInt<U164>; pub type N164 = NInt<U164>;
pub type U165 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P165 = PInt<U165>; pub type N165 = NInt<U165>;
pub type U166 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P166 = PInt<U166>; pub type N166 = NInt<U166>;
pub type U167 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P167 = PInt<U167>; pub type N167 = NInt<U167>;
pub type U168 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P168 = PInt<U168>; pub type N168 = NInt<U168>;
pub type U169 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P169 = PInt<U169>; pub type N169 = NInt<U169>;
pub type U170 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P170 = PInt<U170>; pub type N170 = NInt<U170>;
pub type U171 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P171 = PInt<U171>; pub type N171 = NInt<U171>;
pub type U172 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P172 = PInt<U172>; pub type N172 = NInt<U172>;
pub type U173 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P173 = PInt<U173>; pub type N173 = NInt<U173>;
pub type U174 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P174 = PInt<U174>; pub type N174 = NInt<U174>;
pub type U175 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P175 = PInt<U175>; pub type N175 = NInt<U175>;
pub type U176 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
```

```
pub type P176 = PInt<U176>; pub type N176 = NInt<U176>;
pub type U177 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P177 = PInt<U177>; pub type N177 = NInt<U177>;
pub type U178 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P178 = PInt<U178>; pub type N178 = NInt<U178>;
pub type U179 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P179 = PInt<U179>; pub type N179 = NInt<U179>;
pub type U180 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P180 = PInt<U180>; pub type N180 = NInt<U180>;
pub type U181 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P181 = PInt<U181>; pub type N181 = NInt<U181>;
pub type U182 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P182 = PInt<U182>; pub type N182 = NInt<U182>;
pub type U183 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P183 = PInt<U183>; pub type N183 = NInt<U183>;
pub type U184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P184 = PInt<U184>; pub type N184 = NInt<U184>;
pub type U185 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P185 = PInt<U185>; pub type N185 = NInt<U185>;
pub type U186 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P186 = PInt<U186>; pub type N186 = NInt<U186>;
pub type U187 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P187 = PInt<U187>; pub type N187 = NInt<U187>;
pub type U188 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P188 = PInt<U188>; pub type N188 = NInt<U188>;
pub type U189 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P189 = PInt<U189>; pub type N189 = NInt<U189>;
pub type U190 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P190 = PInt<U190>; pub type N190 = NInt<U190>;
pub type U191 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P191 = PInt<U191>; pub type N191 = NInt<U191>;
pub type U192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P192 = PInt<U192>; pub type N192 = NInt<U192>;
pub type U193 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P193 = PInt<U193>; pub type N193 = NInt<U193>;
pub type U194 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P194 = PInt<U194>; pub type N194 = NInt<U194>;
pub type U195 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P195 = PInt<U195>; pub type N195 = NInt<U195>;
pub type U196 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P196 = PInt<U196>; pub type N196 = NInt<U196>;
pub type U197 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P197 = PInt<U197>; pub type N197 = NInt<U197>;
pub type U198 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P198 = PInt<U198>; pub type N198 = NInt<U198>;
pub type U199 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P199 = PInt<U199>; pub type N199 = NInt<U199>;
pub type U200 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P200 = PInt<U200>; pub type N200 = NInt<U200>;
pub type U201 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P201 = PInt<U201>; pub type N201 = NInt<U201>;
pub type U202 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P202 = PInt<U202>; pub type N202 = NInt<U202>;
pub type U203 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P203 = PInt<U203>; pub type N203 = NInt<U203>;
pub type U204 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P204 = PInt<U204>; pub type N204 = NInt<U204>;
pub type U205 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P205 = PInt<U205>; pub type N205 = NInt<U205>;
pub type U206 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P206 = PInt<U206>; pub type N206 = NInt<U206>;
pub type U207 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P207 = PInt<U207>; pub type N207 = NInt<U207>;
pub type U208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P208 = PInt<U208>; pub type N208 = NInt<U208>;
pub type U209 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P209 = PInt<U209>; pub type N209 = NInt<U209>;
pub type U210 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P210 = PInt<U210>; pub type N210 = NInt<U210>;
pub type U211 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P211 = PInt<U211>; pub type N211 = NInt<U211>;
pub type U212 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P212 = PInt<U212>; pub type N212 = NInt<U212>;
pub type U213 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P213 = PInt<U213>; pub type N213 = NInt<U213>;
pub type U214 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P214 = PInt<U214>; pub type N214 = NInt<U214>;
pub type U215 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P215 = PInt<U215>; pub type N215 = NInt<U215>;
pub type U216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P216 = PInt<U216>; pub type N216 = NInt<U216>;
pub type U217 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P217 = PInt<U217>; pub type N217 = NInt<U217>;
pub type U218 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P218 = PInt<U218>; pub type N218 = NInt<U218>;
pub type U219 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P219 = PInt<U219>; pub type N219 = NInt<U219>;
pub type U220 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P220 = PInt<U220>; pub type N220 = NInt<U220>;
pub type U221 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P221 = PInt<U221>; pub type N221 = NInt<U221>;
pub type U222 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P222 = PInt<U222>; pub type N222 = NInt<U222>;
pub type U223 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P223 = PInt<U223>; pub type N223 = NInt<U223>;
pub type U224 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P224 = PInt<U224>; pub type N224 = NInt<U224>;
pub type U225 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P225 = PInt<U225>; pub type N225 = NInt<U225>;
pub type U226 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P226 = PInt<U226>; pub type N226 = NInt<U226>;
pub type U227 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P227 = PInt<U227>; pub type N227 = NInt<U227>;
pub type U228 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P228 = PInt<U228>; pub type N228 = NInt<U228>;
pub type U229 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P229 = PInt<U229>; pub type N229 = NInt<U229>;
pub type U230 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P230 = PInt<U230>; pub type N230 = NInt<U230>;
pub type U231 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P231 = PInt<U231>; pub type N231 = NInt<U231>;
pub type U232 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P232 = PInt<U232>; pub type N232 = NInt<U232>;
pub type U233 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P233 = PInt<U233>; pub type N233 = NInt<U233>;
pub type U234 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P234 = PInt<U234>; pub type N234 = NInt<U234>;
pub type U235 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P235 = PInt<U235>; pub type N235 = NInt<U235>;
pub type U236 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P236 = PInt<U236>; pub type N236 = NInt<U236>;
pub type U237 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P237 = PInt<U237>; pub type N237 = NInt<U237>;
pub type U238 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P238 = PInt<U238>; pub type N238 = NInt<U238>;
pub type U239 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P239 = PInt<U239>; pub type N239 = NInt<U239>;
pub type U240 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P240 = PInt<U240>; pub type N240 = NInt<U240>;
pub type U241 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P241 = PInt<U241>; pub type N241 = NInt<U241>;
pub type U242 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P242 = PInt<U242>; pub type N242 = NInt<U242>;
pub type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P243 = PInt<U243>; pub type N243 = NInt<U243>;
pub type U244 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P244 = PInt<U244>; pub type N244 = NInt<U244>;
pub type U245 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P245 = PInt<U245>; pub type N245 = NInt<U245>;
pub type U246 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P246 = PInt<U246>; pub type N246 = NInt<U246>;
pub type U247 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P247 = PInt<U247>; pub type N247 = NInt<U247>;
pub type U248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P248 = PInt<U248>; pub type N248 = NInt<U248>;
pub type U249 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P249 = PInt<U249>; pub type N249 = NInt<U249>;
pub type U250 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P250 = PInt<U250>; pub type N250 = NInt<U250>;
pub type U251 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P251 = PInt<U251>; pub type N251 = NInt<U251>;
pub type U252 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P252 = PInt<U252>; pub type N252 = NInt<U252>;
pub type U253 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P253 = PInt<U253>; pub type N253 = NInt<U253>;
pub type U254 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
```

```
pub type P254 = PInt<U254>; pub type N254 = NInt<U254>;
pub type U255 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P255 = PInt<U255>; pub type N255 = NInt<U255>;
pub type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P256 = PInt<U256>; pub type N256 = NInt<U256>;
pub type U257 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P257 = PInt<U257>; pub type N257 = NInt<U257>;
pub type U258 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P258 = PInt<U258>; pub type N258 = NInt<U258>;
pub type U259 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P259 = PInt<U259>; pub type N259 = NInt<U259>;
pub type U260 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P260 = PInt<U260>; pub type N260 = NInt<U260>;
pub type U261 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P261 = PInt<U261>; pub type N261 = NInt<U261>;
pub type U262 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P262 = PInt<U262>; pub type N262 = NInt<U262>;
pub type U263 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P263 = PInt<U263>; pub type N263 = NInt<U263>;
pub type U264 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P264 = PInt<U264>; pub type N264 = NInt<U264>;
pub type U265 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P265 = PInt<U265>; pub type N265 = NInt<U265>;
pub type U266 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P266 = PInt<U266>; pub type N266 = NInt<U266>;
pub type U267 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P267 = PInt<U267>; pub type N267 = NInt<U267>;
pub type U268 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P268 = PInt<U268>; pub type N268 = NInt<U268>;
pub type U269 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P269 = PInt<U269>; pub type N269 = NInt<U269>;
pub type U270 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P270 = PInt<U270>; pub type N270 = NInt<U270>;
pub type U271 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P271 = PInt<U271>; pub type N271 = NInt<U271>;
pub type U272 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P272 = PInt<U272>; pub type N272 = NInt<U272>;
pub type U273 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P273 = PInt<U273>; pub type N273 = NInt<U273>;
pub type U274 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P274 = PInt<U274>; pub type N274 = NInt<U274>;
pub type U275 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P275 = PInt<U275>; pub type N275 = NInt<U275>;
pub type U276 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P276 = PInt<U276>; pub type N276 = NInt<U276>;
pub type U277 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P277 = PInt<U277>; pub type N277 = NInt<U277>;
pub type U278 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P278 = PInt<U278>; pub type N278 = NInt<U278>;
pub type U279 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P279 = PInt<U279>; pub type N279 = NInt<U279>;
pub type U280 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P280 = PInt<U280>; pub type N280 = NInt<U280>;
pub type U281 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P281 = PInt<U281>; pub type N281 = NInt<U281>;
pub type U282 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P282 = PInt<U282>; pub type N282 = NInt<U282>;
pub type U283 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P283 = PInt<U283>; pub type N283 = NInt<U283>;
pub type U284 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P284 = PInt<U284>; pub type N284 = NInt<U284>;
pub type U285 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P285 = PInt<U285>; pub type N285 = NInt<U285>;
pub type U286 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P286 = PInt<U286>; pub type N286 = NInt<U286>;
pub type U287 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P287 = PInt<U287>; pub type N287 = NInt<U287>;
pub type U288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P288 = PInt<U288>; pub type N288 = NInt<U288>;
pub type U289 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P289 = PInt<U289>; pub type N289 = NInt<U289>;
pub type U290 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P290 = PInt<U290>; pub type N290 = NInt<U290>;
pub type U291 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P291 = PInt<U291>; pub type N291 = NInt<U291>;
pub type U292 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P292 = PInt<U292>; pub type N292 = NInt<U292>;
pub type U293 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P293 = PInt<U293>; pub type N293 = NInt<U293>;
pub type U294 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P294 = PInt<U294>; pub type N294 = NInt<U294>;
pub type U295 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P295 = PInt<U295>; pub type N295 = NInt<U295>;
pub type U296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P296 = PInt<U296>; pub type N296 = NInt<U296>;
pub type U297 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P297 = PInt<U297>; pub type N297 = NInt<U297>;
pub type U298 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P298 = PInt<U298>; pub type N298 = NInt<U298>;
pub type U299 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P299 = PInt<U299>; pub type N299 = NInt<U299>;
pub type U300 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P300 = PInt<U300>; pub type N300 = NInt<U300>;
pub type U301 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P301 = PInt<U301>; pub type N301 = NInt<U301>;
pub type U302 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P302 = PInt<U302>; pub type N302 = NInt<U302>;
pub type U303 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P303 = PInt<U303>; pub type N303 = NInt<U303>;
pub type U304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P304 = PInt<U304>; pub type N304 = NInt<U304>;
pub type U305 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P305 = PInt<U305>; pub type N305 = NInt<U305>;
pub type U306 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P306 = PInt<U306>; pub type N306 = NInt<U306>;
pub type U307 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P307 = PInt<U307>; pub type N307 = NInt<U307>;
pub type U308 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P308 = PInt<U308>; pub type N308 = NInt<U308>;
pub type U309 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P309 = PInt<U309>; pub type N309 = NInt<U309>;
pub type U310 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P310 = PInt<U310>; pub type N310 = NInt<U310>;
pub type U311 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P311 = PInt<U311>; pub type N311 = NInt<U311>;
pub type U312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P312 = PInt<U312>; pub type N312 = NInt<U312>;
pub type U313 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P313 = PInt<U313>; pub type N313 = NInt<U313>;
pub type U314 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P314 = PInt<U314>; pub type N314 = NInt<U314>;
pub type U315 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P315 = PInt<U315>; pub type N315 = NInt<U315>;
pub type U316 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P316 = PInt<U316>; pub type N316 = NInt<U316>;
pub type U317 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P317 = PInt<U317>; pub type N317 = NInt<U317>;
pub type U318 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P318 = PInt<U318>; pub type N318 = NInt<U318>;
pub type U319 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P319 = PInt<U319>; pub type N319 = NInt<U319>;
pub type U320 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P320 = PInt<U320>; pub type N320 = NInt<U320>;
pub type U321 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P321 = PInt<U321>; pub type N321 = NInt<U321>;
pub type U322 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P322 = PInt<U322>; pub type N322 = NInt<U322>;
pub type U323 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P323 = PInt<U323>; pub type N323 = NInt<U323>;
pub type U324 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P324 = PInt<U324>; pub type N324 = NInt<U324>;
pub type U325 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P325 = PInt<U325>; pub type N325 = NInt<U325>;
pub type U326 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P326 = PInt<U326>; pub type N326 = NInt<U326>;
pub type U327 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P327 = PInt<U327>; pub type N327 = NInt<U327>;
pub type U328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P328 = PInt<U328>; pub type N328 = NInt<U328>;
pub type U329 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P329 = PInt<U329>; pub type N329 = NInt<U329>;
pub type U330 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P330 = PInt<U330>; pub type N330 = NInt<U330>;
pub type U331 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P331 = PInt<U331>; pub type N331 = NInt<U331>;
pub type U332 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P332 = PInt<U332>; pub type N332 = NInt<U332>;
pub type U333 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P333 = PInt<U333>; pub type N333 = NInt<U333>;
pub type U334 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P334 = PInt<U334>; pub type N334 = NInt<U334>;
pub type U335 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P335 = PInt<U335>; pub type N335 = NInt<U335>;
pub type U336 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P336 = PInt<U336>; pub type N336 = NInt<U336>;
pub type U337 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P337 = PInt<U337>; pub type N337 = NInt<U337>;
pub type U338 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P338 = PInt<U338>; pub type N338 = NInt<U338>;
pub type U339 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P339 = PInt<U339>; pub type N339 = NInt<U339>;
pub type U340 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P340 = PInt<U340>; pub type N340 = NInt<U340>;
pub type U341 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P341 = PInt<U341>; pub type N341 = NInt<U341>;
pub type U342 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P342 = PInt<U342>; pub type N342 = NInt<U342>;
pub type U343 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P343 = PInt<U343>; pub type N343 = NInt<U343>;
pub type U344 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P344 = PInt<U344>; pub type N344 = NInt<U344>;
pub type U345 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P345 = PInt<U345>; pub type N345 = NInt<U345>;
pub type U346 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P346 = PInt<U346>; pub type N346 = NInt<U346>;
pub type U347 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P347 = PInt<U347>; pub type N347 = NInt<U347>;
pub type U348 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P348 = PInt<U348>; pub type N348 = NInt<U348>;
pub type U349 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P349 = PInt<U349>; pub type N349 = NInt<U349>;
pub type U350 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P350 = PInt<U350>; pub type N350 = NInt<U350>;
pub type U351 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P351 = PInt<U351>; pub type N351 = NInt<U351>;
pub type U352 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P352 = PInt<U352>; pub type N352 = NInt<U352>;
pub type U353 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P353 = PInt<U353>; pub type N353 = NInt<U353>;
pub type U354 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P354 = PInt<U354>; pub type N354 = NInt<U354>;
pub type U355 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P355 = PInt<U355>; pub type N355 = NInt<U355>;
pub type U356 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P356 = PInt<U356>; pub type N356 = NInt<U356>;
pub type U357 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P357 = PInt<U357>; pub type N357 = NInt<U357>;
pub type U358 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P358 = PInt<U358>; pub type N358 = NInt<U358>;
pub type U359 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P359 = PInt<U359>; pub type N359 = NInt<U359>;
pub type U360 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P360 = PInt<U360>; pub type N360 = NInt<U360>;
pub type U361 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P361 = PInt<U361>; pub type N361 = NInt<U361>;
pub type U362 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P362 = PInt<U362>; pub type N362 = NInt<U362>;
pub type U363 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P363 = PInt<U363>; pub type N363 = NInt<U363>;
pub type U364 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P364 = PInt<U364>; pub type N364 = NInt<U364>;
pub type U365 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P365 = PInt<U365>; pub type N365 = NInt<U365>;
pub type U366 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P366 = PInt<U366>; pub type N366 = NInt<U366>;
pub type U367 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P367 = PInt<U367>; pub type N367 = NInt<U367>;
pub type U368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P368 = PInt<U368>; pub type N368 = NInt<U368>;
pub type U369 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P369 = PInt<U369>; pub type N369 = NInt<U369>;
pub type U370 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P370 = PInt<U370>; pub type N370 = NInt<U370>;
pub type U371 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P371 = PInt<U371>; pub type N371 = NInt<U371>;
pub type U372 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P372 = PInt<U372>; pub type N372 = NInt<U372>;
pub type U373 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P373 = PInt<U373>; pub type N373 = NInt<U373>;
pub type U374 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P374 = PInt<U374>; pub type N374 = NInt<U374>;
pub type U375 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P375 = PInt<U375>; pub type N375 = NInt<U375>;
pub type U376 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P376 = PInt<U376>; pub type N376 = NInt<U376>;
pub type U377 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P377 = PInt<U377>; pub type N377 = NInt<U377>;
pub type U378 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P378 = PInt<U378>; pub type N378 = NInt<U378>;
pub type U379 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P379 = PInt<U379>; pub type N379 = NInt<U379>;
pub type U380 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P380 = PInt<U380>; pub type N380 = NInt<U380>;
pub type U381 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P381 = PInt<U381>; pub type N381 = NInt<U381>;
pub type U382 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P382 = PInt<U382>; pub type N382 = NInt<U382>;
pub type U383 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P383 = PInt<U383>; pub type N383 = NInt<U383>;
pub type U384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P384 = PInt<U384>; pub type N384 = NInt<U384>;
pub type U385 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P385 = PInt<U385>; pub type N385 = NInt<U385>;
pub type U386 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P386 = PInt<U386>; pub type N386 = NInt<U386>;
pub type U387 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P387 = PInt<U387>; pub type N387 = NInt<U387>;
pub type U388 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P388 = PInt<U388>; pub type N388 = NInt<U388>;
pub type U389 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P389 = PInt<U389>; pub type N389 = NInt<U389>;
pub type U390 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P390 = PInt<U390>; pub type N390 = NInt<U390>;
pub type U391 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P391 = PInt<U391>; pub type N391 = NInt<U391>;
pub type U392 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P392 = PInt<U392>; pub type N392 = NInt<U392>;
pub type U393 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P393 = PInt<U393>; pub type N393 = NInt<U393>;
pub type U394 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P394 = PInt<U394>; pub type N394 = NInt<U394>;
pub type U395 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P395 = PInt<U395>; pub type N395 = NInt<U395>;
pub type U396 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P396 = PInt<U396>; pub type N396 = NInt<U396>;
pub type U397 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P397 = PInt<U397>; pub type N397 = NInt<U397>;
pub type U398 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P398 = PInt<U398>; pub type N398 = NInt<U398>;
pub type U399 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P399 = PInt<U399>; pub type N399 = NInt<U399>;
pub type U400 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P400 = PInt<U400>; pub type N400 = NInt<U400>;
pub type U401 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P401 = PInt<U401>; pub type N401 = NInt<U401>;
pub type U402 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P402 = PInt<U402>; pub type N402 = NInt<U402>;
pub type U403 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P403 = PInt<U403>; pub type N403 = NInt<U403>;
pub type U404 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P404 = PInt<U404>; pub type N404 = NInt<U404>;
pub type U405 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P405 = PInt<U405>; pub type N405 = NInt<U405>;
pub type U406 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P406 = PInt<U406>; pub type N406 = NInt<U406>;
pub type U407 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P407 = PInt<U407>; pub type N407 = NInt<U407>;
pub type U408 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P408 = PInt<U408>; pub type N408 = NInt<U408>;
pub type U409 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P409 = PInt<U409>; pub type N409 = NInt<U409>;
pub type U410 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P410 = PInt<U410>; pub type N410 = NInt<U410>;
pub type U411 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P411 = PInt<U411>; pub type N411 = NInt<U411>;
pub type U412 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P412 = PInt<U412>; pub type N412 = NInt<U412>;
pub type U413 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P413 = PInt<U413>; pub type N413 = NInt<U413>;
pub type U414 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P414 = PInt<U414>; pub type N414 = NInt<U414>;
pub type U415 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P415 = PInt<U415>; pub type N415 = NInt<U415>;
pub type U416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P416 = PInt<U416>; pub type N416 = NInt<U416>;
pub type U417 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P417 = PInt<U417>; pub type N417 = NInt<U417>;
pub type U418 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P418 = PInt<U418>; pub type N418 = NInt<U418>;
pub type U419 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P419 = PInt<U419>; pub type N419 = NInt<U419>;
pub type U420 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P420 = PInt<U420>; pub type N420 = NInt<U420>;
pub type U421 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P421 = PInt<U421>; pub type N421 = NInt<U421>;
pub type U422 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P422 = PInt<U422>; pub type N422 = NInt<U422>;
pub type U423 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P423 = PInt<U423>; pub type N423 = NInt<U423>;
pub type U424 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P424 = PInt<U424>; pub type N424 = NInt<U424>;
pub type U425 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P425 = PInt<U425>; pub type N425 = NInt<U425>;
pub type U426 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P426 = PInt<U426>; pub type N426 = NInt<U426>;
pub type U427 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P427 = PInt<U427>; pub type N427 = NInt<U427>;
pub type U428 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P428 = PInt<U428>; pub type N428 = NInt<U428>;
pub type U429 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P429 = PInt<U429>; pub type N429 = NInt<U429>;
pub type U430 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P430 = PInt<U430>; pub type N430 = NInt<U430>;
pub type U431 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P431 = PInt<U431>; pub type N431 = NInt<U431>;
pub type U432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P432 = PInt<U432>; pub type N432 = NInt<U432>;
pub type U433 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P433 = PInt<U433>; pub type N433 = NInt<U433>;
pub type U434 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P434 = PInt<U434>; pub type N434 = NInt<U434>;
pub type U435 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P435 = PInt<U435>; pub type N435 = NInt<U435>;
pub type U436 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P436 = PInt<U436>; pub type N436 = NInt<U436>;
pub type U437 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P437 = PInt<U437>; pub type N437 = NInt<U437>;
pub type U438 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P438 = PInt<U438>; pub type N438 = NInt<U438>;
pub type U439 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P439 = PInt<U439>; pub type N439 = NInt<U439>;
pub type U440 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P440 = PInt<U440>; pub type N440 = NInt<U440>;
pub type U441 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P441 = PInt<U441>; pub type N441 = NInt<U441>;
pub type U442 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P442 = PInt<U442>; pub type N442 = NInt<U442>;
pub type U443 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P443 = PInt<U443>; pub type N443 = NInt<U443>;
pub type U444 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P444 = PInt<U444>; pub type N444 = NInt<U444>;
pub type U445 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P445 = PInt<U445>; pub type N445 = NInt<U445>;
pub type U446 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P446 = PInt<U446>; pub type N446 = NInt<U446>;
pub type U447 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P447 = PInt<U447>; pub type N447 = NInt<U447>;
pub type U448 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P448 = PInt<U448>; pub type N448 = NInt<U448>;
pub type U449 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P449 = PInt<U449>; pub type N449 = NInt<U449>;
pub type U450 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P450 = PInt<U450>; pub type N450 = NInt<U450>;
pub type U451 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P451 = PInt<U451>; pub type N451 = NInt<U451>;
pub type U452 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P452 = PInt<U452>; pub type N452 = NInt<U452>;
pub type U453 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P453 = PInt<U453>; pub type N453 = NInt<U453>;
pub type U454 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P454 = PInt<U454>; pub type N454 = NInt<U454>;
pub type U455 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P455 = PInt<U455>; pub type N455 = NInt<U455>;
pub type U456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P456 = PInt<U456>; pub type N456 = NInt<U456>;
pub type U457 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P457 = PInt<U457>; pub type N457 = NInt<U457>;
pub type U458 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P458 = PInt<U458>; pub type N458 = NInt<U458>;
pub type U459 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P459 = PInt<U459>; pub type N459 = NInt<U459>;
pub type U460 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P460 = PInt<U460>; pub type N460 = NInt<U460>;
pub type U461 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P461 = PInt<U461>; pub type N461 = NInt<U461>;
pub type U462 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P462 = PInt<U462>; pub type N462 = NInt<U462>;
pub type U463 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P463 = PInt<U463>; pub type N463 = NInt<U463>;
pub type U464 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P464 = PInt<U464>; pub type N464 = NInt<U464>;
pub type U465 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P465 = PInt<U465>; pub type N465 = NInt<U465>;
pub type U466 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P466 = PInt<U466>; pub type N466 = NInt<U466>;
pub type U467 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P467 = PInt<U467>; pub type N467 = NInt<U467>;
pub type U468 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P468 = PInt<U468>; pub type N468 = NInt<U468>;
pub type U469 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P469 = PInt<U469>; pub type N469 = NInt<U469>;
pub type U470 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P470 = PInt<U470>; pub type N470 = NInt<U470>;
pub type U471 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P471 = PInt<U471>; pub type N471 = NInt<U471>;
pub type U472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P472 = PInt<U472>; pub type N472 = NInt<U472>;
pub type U473 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P473 = PInt<U473>; pub type N473 = NInt<U473>;
pub type U474 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P474 = PInt<U474>; pub type N474 = NInt<U474>;
pub type U475 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P475 = PInt<U475>; pub type N475 = NInt<U475>;
pub type U476 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P476 = PInt<U476>; pub type N476 = NInt<U476>;
pub type U477 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P477 = PInt<U477>; pub type N477 = NInt<U477>;
pub type U478 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P478 = PInt<U478>; pub type N478 = NInt<U478>;
pub type U479 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P479 = PInt<U479>; pub type N479 = NInt<U479>;
pub type U480 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P480 = PInt<U480>; pub type N480 = NInt<U480>;
pub type U481 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P481 = PInt<U481>; pub type N481 = NInt<U481>;
pub type U482 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P482 = PInt<U482>; pub type N482 = NInt<U482>;
pub type U483 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P483 = PInt<U483>; pub type N483 = NInt<U483>;
pub type U484 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P484 = PInt<U484>; pub type N484 = NInt<U484>;
pub type U485 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P485 = PInt<U485>; pub type N485 = NInt<U485>;
pub type U486 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P486 = PInt<U486>; pub type N486 = NInt<U486>;
pub type U487 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P487 = PInt<U487>; pub type N487 = NInt<U487>;
pub type U488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
```

```
pub type P488 = PInt<U488>; pub type N488 = NInt<U488>;
pub type U489 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P489 = PInt<U489>; pub type N489 = NInt<U489>;
pub type U490 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P490 = PInt<U490>; pub type N490 = NInt<U490>;
pub type U491 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P491 = PInt<U491>; pub type N491 = NInt<U491>;
pub type U492 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P492 = PInt<U492>; pub type N492 = NInt<U492>;
pub type U493 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P493 = PInt<U493>; pub type N493 = NInt<U493>;
pub type U494 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P494 = PInt<U494>; pub type N494 = NInt<U494>;
pub type U495 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P495 = PInt<U495>; pub type N495 = NInt<U495>;
pub type U496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P496 = PInt<U496>; pub type N496 = NInt<U496>;
pub type U497 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P497 = PInt<U497>; pub type N497 = NInt<U497>;
pub type U498 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P498 = PInt<U498>; pub type N498 = NInt<U498>;
pub type U499 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P499 = PInt<U499>; pub type N499 = NInt<U499>;
pub type U500 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P500 = PInt<U500>; pub type N500 = NInt<U500>;
pub type U501 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P501 = PInt<U501>; pub type N501 = NInt<U501>;
pub type U502 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P502 = PInt<U502>; pub type N502 = NInt<U502>;
pub type U503 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P503 = PInt<U503>; pub type N503 = NInt<U503>;
pub type U504 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P504 = PInt<U504>; pub type N504 = NInt<U504>;
pub type U505 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P505 = PInt<U505>; pub type N505 = NInt<U505>;
pub type U506 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P506 = PInt<U506>; pub type N506 = NInt<U506>;
pub type U507 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P507 = PInt<U507>; pub type N507 = NInt<U507>;
pub type U508 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P508 = PInt<U508>; pub type N508 = NInt<U508>;
pub type U509 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P509 = PInt<U509>; pub type N509 = NInt<U509>;
pub type U510 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P510 = PInt<U510>; pub type N510 = NInt<U510>;
pub type U511 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P511 = PInt<U511>; pub type N511 = NInt<U511>;
pub type U512 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P512 = PInt<U512>; pub type N512 = NInt<U512>;
pub type U513 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P513 = PInt<U513>; pub type N513 = NInt<U513>;
pub type U514 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P514 = PInt<U514>; pub type N514 = NInt<U514>;
pub type U515 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P515 = PInt<U515>; pub type N515 = NInt<U515>;
pub type U516 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P516 = PInt<U516>; pub type N516 = NInt<U516>;
pub type U517 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P517 = PInt<U517>; pub type N517 = NInt<U517>;
pub type U518 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P518 = PInt<U518>; pub type N518 = NInt<U518>;
pub type U519 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P519 = PInt<U519>; pub type N519 = NInt<U519>;
pub type U520 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P520 = PInt<U520>; pub type N520 = NInt<U520>;
pub type U521 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P521 = PInt<U521>; pub type N521 = NInt<U521>;
pub type U522 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P522 = PInt<U522>; pub type N522 = NInt<U522>;
pub type U523 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P523 = PInt<U523>; pub type N523 = NInt<U523>;
pub type U524 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P524 = PInt<U524>; pub type N524 = NInt<U524>;
pub type U525 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P525 = PInt<U525>; pub type N525 = NInt<U525>;
pub type U526 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P526 = PInt<U526>; pub type N526 = NInt<U526>;
pub type U527 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P527 = PInt<U527>; pub type N527 = NInt<U527>;
pub type U528 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P528 = PInt<U528>; pub type N528 = NInt<U528>;
pub type U529 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P529 = PInt<U529>; pub type N529 = NInt<U529>;
pub type U530 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P530 = PInt<U530>; pub type N530 = NInt<U530>;
pub type U531 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P531 = PInt<U531>; pub type N531 = NInt<U531>;
pub type U532 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P532 = PInt<U532>; pub type N532 = NInt<U532>;
pub type U533 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P533 = PInt<U533>; pub type N533 = NInt<U533>;
pub type U534 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P534 = PInt<U534>; pub type N534 = NInt<U534>;
pub type U535 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P535 = PInt<U535>; pub type N535 = NInt<U535>;
pub type U536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P536 = PInt<U536>; pub type N536 = NInt<U536>;
pub type U537 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P537 = PInt<U537>; pub type N537 = NInt<U537>;
pub type U538 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P538 = PInt<U538>; pub type N538 = NInt<U538>;
pub type U539 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P539 = PInt<U539>; pub type N539 = NInt<U539>;
pub type U540 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P540 = PInt<U540>; pub type N540 = NInt<U540>;
pub type U541 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P541 = PInt<U541>; pub type N541 = NInt<U541>;
pub type U542 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P542 = PInt<U542>; pub type N542 = NInt<U542>;
pub type U543 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P543 = PInt<U543>; pub type N543 = NInt<U543>;
pub type U544 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P544 = PInt<U544>; pub type N544 = NInt<U544>;
pub type U545 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P545 = PInt<U545>; pub type N545 = NInt<U545>;
pub type U546 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P546 = PInt<U546>; pub type N546 = NInt<U546>;
pub type U547 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P547 = PInt<U547>; pub type N547 = NInt<U547>;
pub type U548 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P548 = PInt<U548>; pub type N548 = NInt<U548>;
pub type U549 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P549 = PInt<U549>; pub type N549 = NInt<U549>;
pub type U550 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P550 = PInt<U550>; pub type N550 = NInt<U550>;
pub type U551 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P551 = PInt<U551>; pub type N551 = NInt<U551>;
pub type U552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P552 = PInt<U552>; pub type N552 = NInt<U552>;
pub type U553 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P553 = PInt<U553>; pub type N553 = NInt<U553>;
pub type U554 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P554 = PInt<U554>; pub type N554 = NInt<U554>;
pub type U555 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P555 = PInt<U555>; pub type N555 = NInt<U555>;
pub type U556 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P556 = PInt<U556>; pub type N556 = NInt<U556>;
pub type U557 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P557 = PInt<U557>; pub type N557 = NInt<U557>;
pub type U558 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P558 = PInt<U558>; pub type N558 = NInt<U558>;
pub type U559 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P559 = PInt<U559>; pub type N559 = NInt<U559>;
pub type U560 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P560 = PInt<U560>; pub type N560 = NInt<U560>;
pub type U561 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P561 = PInt<U561>; pub type N561 = NInt<U561>;
pub type U562 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P562 = PInt<U562>; pub type N562 = NInt<U562>;
pub type U563 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P563 = PInt<U563>; pub type N563 = NInt<U563>;
pub type U564 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P564 = PInt<U564>; pub type N564 = NInt<U564>;
pub type U565 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P565 = PInt<U565>; pub type N565 = NInt<U565>;
pub type U566 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P566 = PInt<U566>; pub type N566 = NInt<U566>;
pub type U567 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P567 = PInt<U567>; pub type N567 = NInt<U567>;
pub type U568 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P568 = PInt<U568>; pub type N568 = NInt<U568>;
pub type U569 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P569 = PInt<U569>; pub type N569 = NInt<U569>;
pub type U570 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P570 = PInt<U570>; pub type N570 = NInt<U570>;
pub type U571 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P571 = PInt<U571>; pub type N571 = NInt<U571>;
pub type U572 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P572 = PInt<U572>; pub type N572 = NInt<U572>;
pub type U573 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P573 = PInt<U573>; pub type N573 = NInt<U573>;
pub type U574 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P574 = PInt<U574>; pub type N574 = NInt<U574>;
pub type U575 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P575 = PInt<U575>; pub type N575 = NInt<U575>;
pub type U576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P576 = PInt<U576>; pub type N576 = NInt<U576>;
pub type U577 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P577 = PInt<U577>; pub type N577 = NInt<U577>;
pub type U578 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P578 = PInt<U578>; pub type N578 = NInt<U578>;
pub type U579 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P579 = PInt<U579>; pub type N579 = NInt<U579>;
pub type U580 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P580 = PInt<U580>; pub type N580 = NInt<U580>;
pub type U581 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P581 = PInt<U581>; pub type N581 = NInt<U581>;
pub type U582 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P582 = PInt<U582>; pub type N582 = NInt<U582>;
pub type U583 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P583 = PInt<U583>; pub type N583 = NInt<U583>;
pub type U584 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P584 = PInt<U584>; pub type N584 = NInt<U584>;
pub type U585 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P585 = PInt<U585>; pub type N585 = NInt<U585>;
pub type U586 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P586 = PInt<U586>; pub type N586 = NInt<U586>;
pub type U587 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P587 = PInt<U587>; pub type N587 = NInt<U587>;
pub type U588 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P588 = PInt<U588>; pub type N588 = NInt<U588>;
pub type U589 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P589 = PInt<U589>; pub type N589 = NInt<U589>;
pub type U590 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P590 = PInt<U590>; pub type N590 = NInt<U590>;
pub type U591 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P591 = PInt<U591>; pub type N591 = NInt<U591>;
pub type U592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P592 = PInt<U592>; pub type N592 = NInt<U592>;
pub type U593 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P593 = PInt<U593>; pub type N593 = NInt<U593>;
pub type U594 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P594 = PInt<U594>; pub type N594 = NInt<U594>;
pub type U595 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P595 = PInt<U595>; pub type N595 = NInt<U595>;
pub type U596 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P596 = PInt<U596>; pub type N596 = NInt<U596>;
pub type U597 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P597 = PInt<U597>; pub type N597 = NInt<U597>;
pub type U598 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P598 = PInt<U598>; pub type N598 = NInt<U598>;
pub type U599 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P599 = PInt<U599>; pub type N599 = NInt<U599>;
pub type U600 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P600 = PInt<U600>; pub type N600 = NInt<U600>;
pub type U601 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P601 = PInt<U601>; pub type N601 = NInt<U601>;
pub type U602 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P602 = PInt<U602>; pub type N602 = NInt<U602>;
pub type U603 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P603 = PInt<U603>; pub type N603 = NInt<U603>;
pub type U604 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P604 = PInt<U604>; pub type N604 = NInt<U604>;
pub type U605 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P605 = PInt<U605>; pub type N605 = NInt<U605>;
pub type U606 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P606 = PInt<U606>; pub type N606 = NInt<U606>;
pub type U607 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P607 = PInt<U607>; pub type N607 = NInt<U607>;
pub type U608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P608 = PInt<U608>; pub type N608 = NInt<U608>;
pub type U609 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P609 = PInt<U609>; pub type N609 = NInt<U609>;
pub type U610 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P610 = PInt<U610>; pub type N610 = NInt<U610>;
pub type U611 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P611 = PInt<U611>; pub type N611 = NInt<U611>;
pub type U612 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P612 = PInt<U612>; pub type N612 = NInt<U612>;
pub type U613 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P613 = PInt<U613>; pub type N613 = NInt<U613>;
pub type U614 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P614 = PInt<U614>; pub type N614 = NInt<U614>;
pub type U615 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P615 = PInt<U615>; pub type N615 = NInt<U615>;
pub type U616 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P616 = PInt<U616>; pub type N616 = NInt<U616>;
pub type U617 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P617 = PInt<U617>; pub type N617 = NInt<U617>;
pub type U618 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P618 = PInt<U618>; pub type N618 = NInt<U618>;
pub type U619 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P619 = PInt<U619>; pub type N619 = NInt<U619>;
pub type U620 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P620 = PInt<U620>; pub type N620 = NInt<U620>;
pub type U621 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P621 = PInt<U621>; pub type N621 = NInt<U621>;
pub type U622 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P622 = PInt<U622>; pub type N622 = NInt<U622>;
pub type U623 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P623 = PInt<U623>; pub type N623 = NInt<U623>;
pub type U624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P624 = PInt<U624>; pub type N624 = NInt<U624>;
pub type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P625 = PInt<U625>; pub type N625 = NInt<U625>;
pub type U626 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P626 = PInt<U626>; pub type N626 = NInt<U626>;
pub type U627 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P627 = PInt<U627>; pub type N627 = NInt<U627>;
pub type U628 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P628 = PInt<U628>; pub type N628 = NInt<U628>;
pub type U629 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P629 = PInt<U629>; pub type N629 = NInt<U629>;
pub type U630 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P630 = PInt<U630>; pub type N630 = NInt<U630>;
pub type U631 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P631 = PInt<U631>; pub type N631 = NInt<U631>;
pub type U632 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P632 = PInt<U632>; pub type N632 = NInt<U632>;
pub type U633 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P633 = PInt<U633>; pub type N633 = NInt<U633>;
pub type U634 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P634 = PInt<U634>; pub type N634 = NInt<U634>;
pub type U635 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P635 = PInt<U635>; pub type N635 = NInt<U635>;
pub type U636 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P636 = PInt<U636>; pub type N636 = NInt<U636>;
pub type U637 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P637 = PInt<U637>; pub type N637 = NInt<U637>;
pub type U638 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P638 = PInt<U638>; pub type N638 = NInt<U638>;
pub type U639 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P639 = PInt<U639>; pub type N639 = NInt<U639>;
pub type U640 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P640 = PInt<U640>; pub type N640 = NInt<U640>;
pub type U641 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P641 = PInt<U641>; pub type N641 = NInt<U641>;
pub type U642 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P642 = PInt<U642>; pub type N642 = NInt<U642>;
pub type U643 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P643 = PInt<U643>; pub type N643 = NInt<U643>;
pub type U644 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P644 = PInt<U644>; pub type N644 = NInt<U644>;
pub type U645 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P645 = PInt<U645>; pub type N645 = NInt<U645>;
pub type U646 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P646 = PInt<U646>; pub type N646 = NInt<U646>;
pub type U647 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P647 = PInt<U647>; pub type N647 = NInt<U647>;
pub type U648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P648 = PInt<U648>; pub type N648 = NInt<U648>;
pub type U649 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P649 = PInt<U649>; pub type N649 = NInt<U649>;
pub type U650 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P650 = PInt<U650>; pub type N650 = NInt<U650>;
pub type U651 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P651 = PInt<U651>; pub type N651 = NInt<U651>;
pub type U652 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P652 = PInt<U652>; pub type N652 = NInt<U652>;
pub type U653 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P653 = PInt<U653>; pub type N653 = NInt<U653>;
pub type U654 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P654 = PInt<U654>; pub type N654 = NInt<U654>;
pub type U655 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P655 = PInt<U655>; pub type N655 = NInt<U655>;
pub type U656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P656 = PInt<U656>; pub type N656 = NInt<U656>;
pub type U657 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P657 = PInt<U657>; pub type N657 = NInt<U657>;
pub type U658 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P658 = PInt<U658>; pub type N658 = NInt<U658>;
pub type U659 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P659 = PInt<U659>; pub type N659 = NInt<U659>;
pub type U660 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P660 = PInt<U660>; pub type N660 = NInt<U660>;
pub type U661 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P661 = PInt<U661>; pub type N661 = NInt<U661>;
pub type U662 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P662 = PInt<U662>; pub type N662 = NInt<U662>;
pub type U663 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P663 = PInt<U663>; pub type N663 = NInt<U663>;
pub type U664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P664 = PInt<U664>; pub type N664 = NInt<U664>;
pub type U665 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P665 = PInt<U665>; pub type N665 = NInt<U665>;
pub type U666 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P666 = PInt<U666>; pub type N666 = NInt<U666>;
pub type U667 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P667 = PInt<U667>; pub type N667 = NInt<U667>;
pub type U668 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P668 = PInt<U668>; pub type N668 = NInt<U668>;
pub type U669 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P669 = PInt<U669>; pub type N669 = NInt<U669>;
pub type U670 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P670 = PInt<U670>; pub type N670 = NInt<U670>;
pub type U671 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P671 = PInt<U671>; pub type N671 = NInt<U671>;
pub type U672 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P672 = PInt<U672>; pub type N672 = NInt<U672>;
pub type U673 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P673 = PInt<U673>; pub type N673 = NInt<U673>;
pub type U674 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P674 = PInt<U674>; pub type N674 = NInt<U674>;
pub type U675 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P675 = PInt<U675>; pub type N675 = NInt<U675>;
pub type U676 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P676 = PInt<U676>; pub type N676 = NInt<U676>;
pub type U677 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P677 = PInt<U677>; pub type N677 = NInt<U677>;
pub type U678 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P678 = PInt<U678>; pub type N678 = NInt<U678>;
pub type U679 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P679 = PInt<U679>; pub type N679 = NInt<U679>;
pub type U680 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P680 = PInt<U680>; pub type N680 = NInt<U680>;
pub type U681 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P681 = PInt<U681>; pub type N681 = NInt<U681>;
pub type U682 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P682 = PInt<U682>; pub type N682 = NInt<U682>;
pub type U683 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P683 = PInt<U683>; pub type N683 = NInt<U683>;
pub type U684 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P684 = PInt<U684>; pub type N684 = NInt<U684>;
pub type U685 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P685 = PInt<U685>; pub type N685 = NInt<U685>;
pub type U686 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P686 = PInt<U686>; pub type N686 = NInt<U686>;
pub type U687 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P687 = PInt<U687>; pub type N687 = NInt<U687>;
pub type U688 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P688 = PInt<U688>; pub type N688 = NInt<U688>;
pub type U689 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P689 = PInt<U689>; pub type N689 = NInt<U689>;
pub type U690 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P690 = PInt<U690>; pub type N690 = NInt<U690>;
pub type U691 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P691 = PInt<U691>; pub type N691 = NInt<U691>;
pub type U692 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P692 = PInt<U692>; pub type N692 = NInt<U692>;
pub type U693 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P693 = PInt<U693>; pub type N693 = NInt<U693>;
pub type U694 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P694 = PInt<U694>; pub type N694 = NInt<U694>;
pub type U695 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P695 = PInt<U695>; pub type N695 = NInt<U695>;
pub type U696 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P696 = PInt<U696>; pub type N696 = NInt<U696>;
pub type U697 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P697 = PInt<U697>; pub type N697 = NInt<U697>;
pub type U698 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P698 = PInt<U698>; pub type N698 = NInt<U698>;
pub type U699 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P699 = PInt<U699>; pub type N699 = NInt<U699>;
pub type U700 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P700 = PInt<U700>; pub type N700 = NInt<U700>;
pub type U701 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P701 = PInt<U701>; pub type N701 = NInt<U701>;
pub type U702 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P702 = PInt<U702>; pub type N702 = NInt<U702>;
pub type U703 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P703 = PInt<U703>; pub type N703 = NInt<U703>;
pub type U704 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P704 = PInt<U704>; pub type N704 = NInt<U704>;
pub type U705 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P705 = PInt<U705>; pub type N705 = NInt<U705>;
pub type U706 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P706 = PInt<U706>; pub type N706 = NInt<U706>;
pub type U707 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P707 = PInt<U707>; pub type N707 = NInt<U707>;
pub type U708 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P708 = PInt<U708>; pub type N708 = NInt<U708>;
pub type U709 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P709 = PInt<U709>; pub type N709 = NInt<U709>;
pub type U710 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P710 = PInt<U710>; pub type N710 = NInt<U710>;
pub type U711 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P711 = PInt<U711>; pub type N711 = NInt<U711>;
pub type U712 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P712 = PInt<U712>; pub type N712 = NInt<U712>;
pub type U713 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P713 = PInt<U713>; pub type N713 = NInt<U713>;
pub type U714 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P714 = PInt<U714>; pub type N714 = NInt<U714>;
pub type U715 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P715 = PInt<U715>; pub type N715 = NInt<U715>;
pub type U716 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P716 = PInt<U716>; pub type N716 = NInt<U716>;
pub type U717 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P717 = PInt<U717>; pub type N717 = NInt<U717>;
pub type U718 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P718 = PInt<U718>; pub type N718 = NInt<U718>;
pub type U719 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P719 = PInt<U719>; pub type N719 = NInt<U719>;
pub type U720 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P720 = PInt<U720>; pub type N720 = NInt<U720>;
pub type U721 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P721 = PInt<U721>; pub type N721 = NInt<U721>;
pub type U722 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P722 = PInt<U722>; pub type N722 = NInt<U722>;
pub type U723 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P723 = PInt<U723>; pub type N723 = NInt<U723>;
pub type U724 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P724 = PInt<U724>; pub type N724 = NInt<U724>;
pub type U725 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P725 = PInt<U725>; pub type N725 = NInt<U725>;
pub type U726 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P726 = PInt<U726>; pub type N726 = NInt<U726>;
pub type U727 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P727 = PInt<U727>; pub type N727 = NInt<U727>;
pub type U728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P728 = PInt<U728>; pub type N728 = NInt<U728>;
pub type U729 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P729 = PInt<U729>; pub type N729 = NInt<U729>;
pub type U730 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P730 = PInt<U730>; pub type N730 = NInt<U730>;
pub type U731 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P731 = PInt<U731>; pub type N731 = NInt<U731>;
pub type U732 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P732 = PInt<U732>; pub type N732 = NInt<U732>;
pub type U733 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P733 = PInt<U733>; pub type N733 = NInt<U733>;
pub type U734 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P734 = PInt<U734>; pub type N734 = NInt<U734>;
pub type U735 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P735 = PInt<U735>; pub type N735 = NInt<U735>;
pub type U736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P736 = PInt<U736>; pub type N736 = NInt<U736>;
pub type U737 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P737 = PInt<U737>; pub type N737 = NInt<U737>;
pub type U738 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P738 = PInt<U738>; pub type N738 = NInt<U738>;
pub type U739 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P739 = PInt<U739>; pub type N739 = NInt<U739>;
pub type U740 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P740 = PInt<U740>; pub type N740 = NInt<U740>;
pub type U741 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P741 = PInt<U741>; pub type N741 = NInt<U741>;
pub type U742 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P742 = PInt<U742>; pub type N742 = NInt<U742>;
pub type U743 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P743 = PInt<U743>; pub type N743 = NInt<U743>;
pub type U744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P744 = PInt<U744>; pub type N744 = NInt<U744>;
pub type U745 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P745 = PInt<U745>; pub type N745 = NInt<U745>;
pub type U746 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P746 = PInt<U746>; pub type N746 = NInt<U746>;
pub type U747 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P747 = PInt<U747>; pub type N747 = NInt<U747>;
pub type U748 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P748 = PInt<U748>; pub type N748 = NInt<U748>;
pub type U749 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P749 = PInt<U749>; pub type N749 = NInt<U749>;
pub type U750 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P750 = PInt<U750>; pub type N750 = NInt<U750>;
pub type U751 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P751 = PInt<U751>; pub type N751 = NInt<U751>;
pub type U752 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P752 = PInt<U752>; pub type N752 = NInt<U752>;
pub type U753 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P753 = PInt<U753>; pub type N753 = NInt<U753>;
pub type U754 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P754 = PInt<U754>; pub type N754 = NInt<U754>;
pub type U755 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P755 = PInt<U755>; pub type N755 = NInt<U755>;
pub type U756 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P756 = PInt<U756>; pub type N756 = NInt<U756>;
pub type U757 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P757 = PInt<U757>; pub type N757 = NInt<U757>;
pub type U758 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P758 = PInt<U758>; pub type N758 = NInt<U758>;
pub type U759 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P759 = PInt<U759>; pub type N759 = NInt<U759>;
pub type U760 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P760 = PInt<U760>; pub type N760 = NInt<U760>;
pub type U761 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P761 = PInt<U761>; pub type N761 = NInt<U761>;
pub type U762 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P762 = PInt<U762>; pub type N762 = NInt<U762>;
pub type U763 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P763 = PInt<U763>; pub type N763 = NInt<U763>;
pub type U764 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P764 = PInt<U764>; pub type N764 = NInt<U764>;
pub type U765 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P765 = PInt<U765>; pub type N765 = NInt<U765>;
pub type U766 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P766 = PInt<U766>; pub type N766 = NInt<U766>;
pub type U767 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P767 = PInt<U767>; pub type N767 = NInt<U767>;
pub type U768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P768 = PInt<U768>; pub type N768 = NInt<U768>;
pub type U769 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P769 = PInt<U769>; pub type N769 = NInt<U769>;
pub type U770 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P770 = PInt<U770>; pub type N770 = NInt<U770>;
pub type U771 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P771 = PInt<U771>; pub type N771 = NInt<U771>;
pub type U772 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P772 = PInt<U772>; pub type N772 = NInt<U772>;
pub type U773 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P773 = PInt<U773>; pub type N773 = NInt<U773>;
pub type U774 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P774 = PInt<U774>; pub type N774 = NInt<U774>;
pub type U775 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P775 = PInt<U775>; pub type N775 = NInt<U775>;
pub type U776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P776 = PInt<U776>; pub type N776 = NInt<U776>;
pub type U777 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P777 = PInt<U777>; pub type N777 = NInt<U777>;
pub type U778 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P778 = PInt<U778>; pub type N778 = NInt<U778>;
pub type U779 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P779 = PInt<U779>; pub type N779 = NInt<U779>;
pub type U780 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P780 = PInt<U780>; pub type N780 = NInt<U780>;
pub type U781 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P781 = PInt<U781>; pub type N781 = NInt<U781>;
pub type U782 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P782 = PInt<U782>; pub type N782 = NInt<U782>;
pub type U783 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P783 = PInt<U783>; pub type N783 = NInt<U783>;
pub type U784 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P784 = PInt<U784>; pub type N784 = NInt<U784>;
pub type U785 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P785 = PInt<U785>; pub type N785 = NInt<U785>;
pub type U786 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P786 = PInt<U786>; pub type N786 = NInt<U786>;
pub type U787 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P787 = PInt<U787>; pub type N787 = NInt<U787>;
pub type U788 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P788 = PInt<U788>; pub type N788 = NInt<U788>;
pub type U789 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P789 = PInt<U789>; pub type N789 = NInt<U789>;
pub type U790 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P790 = PInt<U790>; pub type N790 = NInt<U790>;
pub type U791 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P791 = PInt<U791>; pub type N791 = NInt<U791>;
pub type U792 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P792 = PInt<U792>; pub type N792 = NInt<U792>;
pub type U793 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P793 = PInt<U793>; pub type N793 = NInt<U793>;
pub type U794 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P794 = PInt<U794>; pub type N794 = NInt<U794>;
pub type U795 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P795 = PInt<U795>; pub type N795 = NInt<U795>;
pub type U796 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P796 = PInt<U796>; pub type N796 = NInt<U796>;
pub type U797 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P797 = PInt<U797>; pub type N797 = NInt<U797>;
pub type U798 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P798 = PInt<U798>; pub type N798 = NInt<U798>;
pub type U799 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P799 = PInt<U799>; pub type N799 = NInt<U799>;
pub type U800 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P800 = PInt<U800>; pub type N800 = NInt<U800>;
pub type U801 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P801 = PInt<U801>; pub type N801 = NInt<U801>;
pub type U802 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P802 = PInt<U802>; pub type N802 = NInt<U802>;
pub type U803 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P803 = PInt<U803>; pub type N803 = NInt<U803>;
pub type U804 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P804 = PInt<U804>; pub type N804 = NInt<U804>;
pub type U805 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P805 = PInt<U805>; pub type N805 = NInt<U805>;
pub type U806 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P806 = PInt<U806>; pub type N806 = NInt<U806>;
pub type U807 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P807 = PInt<U807>; pub type N807 = NInt<U807>;
pub type U808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P808 = PInt<U808>; pub type N808 = NInt<U808>;
pub type U809 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P809 = PInt<U809>; pub type N809 = NInt<U809>;
pub type U810 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P810 = PInt<U810>; pub type N810 = NInt<U810>;
pub type U811 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P811 = PInt<U811>; pub type N811 = NInt<U811>;
pub type U812 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P812 = PInt<U812>; pub type N812 = NInt<U812>;
pub type U813 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P813 = PInt<U813>; pub type N813 = NInt<U813>;
pub type U814 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P814 = PInt<U814>; pub type N814 = NInt<U814>;
pub type U815 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P815 = PInt<U815>; pub type N815 = NInt<U815>;
pub type U816 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P816 = PInt<U816>; pub type N816 = NInt<U816>;
pub type U817 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P817 = PInt<U817>; pub type N817 = NInt<U817>;
pub type U818 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P818 = PInt<U818>; pub type N818 = NInt<U818>;
pub type U819 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P819 = PInt<U819>; pub type N819 = NInt<U819>;
pub type U820 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P820 = PInt<U820>; pub type N820 = NInt<U820>;
pub type U821 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P821 = PInt<U821>; pub type N821 = NInt<U821>;
pub type U822 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P822 = PInt<U822>; pub type N822 = NInt<U822>;
pub type U823 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P823 = PInt<U823>; pub type N823 = NInt<U823>;
pub type U824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P824 = PInt<U824>; pub type N824 = NInt<U824>;
pub type U825 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P825 = PInt<U825>; pub type N825 = NInt<U825>;
pub type U826 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P826 = PInt<U826>; pub type N826 = NInt<U826>;
pub type U827 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P827 = PInt<U827>; pub type N827 = NInt<U827>;
pub type U828 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P828 = PInt<U828>; pub type N828 = NInt<U828>;
pub type U829 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P829 = PInt<U829>; pub type N829 = NInt<U829>;
pub type U830 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P830 = PInt<U830>; pub type N830 = NInt<U830>;
pub type U831 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P831 = PInt<U831>; pub type N831 = NInt<U831>;
pub type U832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P832 = PInt<U832>; pub type N832 = NInt<U832>;
pub type U833 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P833 = PInt<U833>; pub type N833 = NInt<U833>;
pub type U834 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P834 = PInt<U834>; pub type N834 = NInt<U834>;
pub type U835 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P835 = PInt<U835>; pub type N835 = NInt<U835>;
pub type U836 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P836 = PInt<U836>; pub type N836 = NInt<U836>;
pub type U837 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P837 = PInt<U837>; pub type N837 = NInt<U837>;
pub type U838 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P838 = PInt<U838>; pub type N838 = NInt<U838>;
pub type U839 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P839 = PInt<U839>; pub type N839 = NInt<U839>;
pub type U840 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P840 = PInt<U840>; pub type N840 = NInt<U840>;
pub type U841 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P841 = PInt<U841>; pub type N841 = NInt<U841>;
pub type U842 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P842 = PInt<U842>; pub type N842 = NInt<U842>;
pub type U843 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P843 = PInt<U843>; pub type N843 = NInt<U843>;
pub type U844 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P844 = PInt<U844>; pub type N844 = NInt<U844>;
pub type U845 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P845 = PInt<U845>; pub type N845 = NInt<U845>;
pub type U846 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P846 = PInt<U846>; pub type N846 = NInt<U846>;
pub type U847 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P847 = PInt<U847>; pub type N847 = NInt<U847>;
pub type U848 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P848 = PInt<U848>; pub type N848 = NInt<U848>;
pub type U849 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P849 = PInt<U849>; pub type N849 = NInt<U849>;
pub type U850 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P850 = PInt<U850>; pub type N850 = NInt<U850>;
pub type U851 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P851 = PInt<U851>; pub type N851 = NInt<U851>;
pub type U852 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P852 = PInt<U852>; pub type N852 = NInt<U852>;
pub type U853 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P853 = PInt<U853>; pub type N853 = NInt<U853>;
pub type U854 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P854 = PInt<U854>; pub type N854 = NInt<U854>;
pub type U855 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P855 = PInt<U855>; pub type N855 = NInt<U855>;
pub type U856 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P856 = PInt<U856>; pub type N856 = NInt<U856>;
pub type U857 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P857 = PInt<U857>; pub type N857 = NInt<U857>;
pub type U858 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P858 = PInt<U858>; pub type N858 = NInt<U858>;
pub type U859 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P859 = PInt<U859>; pub type N859 = NInt<U859>;
pub type U860 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P860 = PInt<U860>; pub type N860 = NInt<U860>;
pub type U861 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P861 = PInt<U861>; pub type N861 = NInt<U861>;
pub type U862 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P862 = PInt<U862>; pub type N862 = NInt<U862>;
pub type U863 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P863 = PInt<U863>; pub type N863 = NInt<U863>;
pub type U864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P864 = PInt<U864>; pub type N864 = NInt<U864>;
pub type U865 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P865 = PInt<U865>; pub type N865 = NInt<U865>;
pub type U866 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P866 = PInt<U866>; pub type N866 = NInt<U866>;
pub type U867 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P867 = PInt<U867>; pub type N867 = NInt<U867>;
pub type U868 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P868 = PInt<U868>; pub type N868 = NInt<U868>;
pub type U869 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P869 = PInt<U869>; pub type N869 = NInt<U869>;
pub type U870 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P870 = PInt<U870>; pub type N870 = NInt<U870>;
pub type U871 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P871 = PInt<U871>; pub type N871 = NInt<U871>;
pub type U872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P872 = PInt<U872>; pub type N872 = NInt<U872>;
pub type U873 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P873 = PInt<U873>; pub type N873 = NInt<U873>;
pub type U874 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P874 = PInt<U874>; pub type N874 = NInt<U874>;
pub type U875 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P875 = PInt<U875>; pub type N875 = NInt<U875>;
pub type U876 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P876 = PInt<U876>; pub type N876 = NInt<U876>;
pub type U877 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P877 = PInt<U877>; pub type N877 = NInt<U877>;
pub type U878 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P878 = PInt<U878>; pub type N878 = NInt<U878>;
pub type U879 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P879 = PInt<U879>; pub type N879 = NInt<U879>;
pub type U880 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P880 = PInt<U880>; pub type N880 = NInt<U880>;
pub type U881 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P881 = PInt<U881>; pub type N881 = NInt<U881>;
pub type U882 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P882 = PInt<U882>; pub type N882 = NInt<U882>;
pub type U883 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P883 = PInt<U883>; pub type N883 = NInt<U883>;
pub type U884 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P884 = PInt<U884>; pub type N884 = NInt<U884>;
pub type U885 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P885 = PInt<U885>; pub type N885 = NInt<U885>;
pub type U886 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P886 = PInt<U886>; pub type N886 = NInt<U886>;
pub type U887 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P887 = PInt<U887>; pub type N887 = NInt<U887>;
pub type U888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P888 = PInt<U888>; pub type N888 = NInt<U888>;
pub type U889 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P889 = PInt<U889>; pub type N889 = NInt<U889>;
pub type U890 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P890 = PInt<U890>; pub type N890 = NInt<U890>;
pub type U891 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P891 = PInt<U891>; pub type N891 = NInt<U891>;
pub type U892 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P892 = PInt<U892>; pub type N892 = NInt<U892>;
pub type U893 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P893 = PInt<U893>; pub type N893 = NInt<U893>;
pub type U894 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P894 = PInt<U894>; pub type N894 = NInt<U894>;
pub type U895 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P895 = PInt<U895>; pub type N895 = NInt<U895>;
pub type U896 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P896 = PInt<U896>; pub type N896 = NInt<U896>;
pub type U897 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P897 = PInt<U897>; pub type N897 = NInt<U897>;
pub type U898 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P898 = PInt<U898>; pub type N898 = NInt<U898>;
pub type U899 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P899 = PInt<U899>; pub type N899 = NInt<U899>;
pub type U900 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P900 = PInt<U900>; pub type N900 = NInt<U900>;
pub type U901 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P901 = PInt<U901>; pub type N901 = NInt<U901>;
pub type U902 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P902 = PInt<U902>; pub type N902 = NInt<U902>;
pub type U903 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P903 = PInt<U903>; pub type N903 = NInt<U903>;
pub type U904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P904 = PInt<U904>; pub type N904 = NInt<U904>;
pub type U905 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P905 = PInt<U905>; pub type N905 = NInt<U905>;
pub type U906 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P906 = PInt<U906>; pub type N906 = NInt<U906>;
pub type U907 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P907 = PInt<U907>; pub type N907 = NInt<U907>;
pub type U908 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P908 = PInt<U908>; pub type N908 = NInt<U908>;
pub type U909 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P909 = PInt<U909>; pub type N909 = NInt<U909>;
pub type U910 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P910 = PInt<U910>; pub type N910 = NInt<U910>;
pub type U911 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P911 = PInt<U911>; pub type N911 = NInt<U911>;
pub type U912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P912 = PInt<U912>; pub type N912 = NInt<U912>;
pub type U913 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P913 = PInt<U913>; pub type N913 = NInt<U913>;
pub type U914 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P914 = PInt<U914>; pub type N914 = NInt<U914>;
pub type U915 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P915 = PInt<U915>; pub type N915 = NInt<U915>;
pub type U916 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P916 = PInt<U916>; pub type N916 = NInt<U916>;
pub type U917 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P917 = PInt<U917>; pub type N917 = NInt<U917>;
pub type U918 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P918 = PInt<U918>; pub type N918 = NInt<U918>;
pub type U919 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P919 = PInt<U919>; pub type N919 = NInt<U919>;
pub type U920 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P920 = PInt<U920>; pub type N920 = NInt<U920>;
pub type U921 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P921 = PInt<U921>; pub type N921 = NInt<U921>;
pub type U922 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P922 = PInt<U922>; pub type N922 = NInt<U922>;
pub type U923 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P923 = PInt<U923>; pub type N923 = NInt<U923>;
pub type U924 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P924 = PInt<U924>; pub type N924 = NInt<U924>;
pub type U925 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P925 = PInt<U925>; pub type N925 = NInt<U925>;
pub type U926 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P926 = PInt<U926>; pub type N926 = NInt<U926>;
pub type U927 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P927 = PInt<U927>; pub type N927 = NInt<U927>;
pub type U928 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P928 = PInt<U928>; pub type N928 = NInt<U928>;
pub type U929 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P929 = PInt<U929>; pub type N929 = NInt<U929>;
pub type U930 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P930 = PInt<U930>; pub type N930 = NInt<U930>;
pub type U931 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P931 = PInt<U931>; pub type N931 = NInt<U931>;
pub type U932 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P932 = PInt<U932>; pub type N932 = NInt<U932>;
pub type U933 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P933 = PInt<U933>; pub type N933 = NInt<U933>;
pub type U934 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P934 = PInt<U934>; pub type N934 = NInt<U934>;
pub type U935 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P935 = PInt<U935>; pub type N935 = NInt<U935>;
pub type U936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P936 = PInt<U936>; pub type N936 = NInt<U936>;
pub type U937 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P937 = PInt<U937>; pub type N937 = NInt<U937>;
pub type U938 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P938 = PInt<U938>; pub type N938 = NInt<U938>;
pub type U939 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P939 = PInt<U939>; pub type N939 = NInt<U939>;
pub type U940 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P940 = PInt<U940>; pub type N940 = NInt<U940>;
pub type U941 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P941 = PInt<U941>; pub type N941 = NInt<U941>;
pub type U942 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P942 = PInt<U942>; pub type N942 = NInt<U942>;
pub type U943 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P943 = PInt<U943>; pub type N943 = NInt<U943>;
pub type U944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P944 = PInt<U944>; pub type N944 = NInt<U944>;
pub type U945 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P945 = PInt<U945>; pub type N945 = NInt<U945>;
pub type U946 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P946 = PInt<U946>; pub type N946 = NInt<U946>;
pub type U947 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P947 = PInt<U947>; pub type N947 = NInt<U947>;
pub type U948 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P948 = PInt<U948>; pub type N948 = NInt<U948>;
pub type U949 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P949 = PInt<U949>; pub type N949 = NInt<U949>;
pub type U950 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P950 = PInt<U950>; pub type N950 = NInt<U950>;
pub type U951 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P951 = PInt<U951>; pub type N951 = NInt<U951>;
pub type U952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P952 = PInt<U952>; pub type N952 = NInt<U952>;
pub type U953 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P953 = PInt<U953>; pub type N953 = NInt<U953>;
pub type U954 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P954 = PInt<U954>; pub type N954 = NInt<U954>;
pub type U955 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P955 = PInt<U955>; pub type N955 = NInt<U955>;
pub type U956 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P956 = PInt<U956>; pub type N956 = NInt<U956>;
pub type U957 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P957 = PInt<U957>; pub type N957 = NInt<U957>;
pub type U958 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P958 = PInt<U958>; pub type N958 = NInt<U958>;
pub type U959 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P959 = PInt<U959>; pub type N959 = NInt<U959>;
pub type U960 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P960 = PInt<U960>; pub type N960 = NInt<U960>;
pub type U961 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P961 = PInt<U961>; pub type N961 = NInt<U961>;
pub type U962 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P962 = PInt<U962>; pub type N962 = NInt<U962>;
pub type U963 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P963 = PInt<U963>; pub type N963 = NInt<U963>;
pub type U964 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P964 = PInt<U964>; pub type N964 = NInt<U964>;
pub type U965 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P965 = PInt<U965>; pub type N965 = NInt<U965>;
pub type U966 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P966 = PInt<U966>; pub type N966 = NInt<U966>;
pub type U967 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P967 = PInt<U967>; pub type N967 = NInt<U967>;
pub type U968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P968 = PInt<U968>; pub type N968 = NInt<U968>;
pub type U969 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P969 = PInt<U969>; pub type N969 = NInt<U969>;
pub type U970 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P970 = PInt<U970>; pub type N970 = NInt<U970>;
pub type U971 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P971 = PInt<U971>; pub type N971 = NInt<U971>;
pub type U972 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P972 = PInt<U972>; pub type N972 = NInt<U972>;
pub type U973 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P973 = PInt<U973>; pub type N973 = NInt<U973>;
pub type U974 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P974 = PInt<U974>; pub type N974 = NInt<U974>;
pub type U975 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P975 = PInt<U975>; pub type N975 = NInt<U975>;
pub type U976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P976 = PInt<U976>; pub type N976 = NInt<U976>;
pub type U977 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P977 = PInt<U977>; pub type N977 = NInt<U977>;
pub type U978 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P978 = PInt<U978>; pub type N978 = NInt<U978>;
pub type U979 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P979 = PInt<U979>; pub type N979 = NInt<U979>;
pub type U980 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P980 = PInt<U980>; pub type N980 = NInt<U980>;
pub type U981 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P981 = PInt<U981>; pub type N981 = NInt<U981>;
pub type U982 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
```

```
pub type P982 = PInt<U982>; pub type N982 = NInt<U982>;
pub type U983 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P983 = PInt<U983>; pub type N983 = NInt<U983>;
pub type U984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P984 = PInt<U984>; pub type N984 = NInt<U984>;
pub type U985 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P985 = PInt<U985>; pub type N985 = NInt<U985>;
pub type U986 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P986 = PInt<U986>; pub type N986 = NInt<U986>;
pub type U987 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P987 = PInt<U987>; pub type N987 = NInt<U987>;
pub type U988 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P988 = PInt<U988>; pub type N988 = NInt<U988>;
pub type U989 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P989 = PInt<U989>; pub type N989 = NInt<U989>;
pub type U990 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P990 = PInt<U990>; pub type N990 = NInt<U990>;
pub type U991 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P991 = PInt<U991>; pub type N991 = NInt<U991>;
pub type U992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P992 = PInt<U992>; pub type N992 = NInt<U992>;
pub type U993 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P993 = PInt<U993>; pub type N993 = NInt<U993>;
pub type U994 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P994 = PInt<U994>; pub type N994 = NInt<U994>;
pub type U995 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P995 = PInt<U995>; pub type N995 = NInt<U995>;
pub type U996 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P996 = PInt<U996>; pub type N996 = NInt<U996>;
pub type U997 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P997 = PInt<U997>; pub type N997 = NInt<U997>;
pub type U998 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P998 = PInt<U998>; pub type N998 = NInt<U998>;
pub type U999 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P999 = PInt<U999>; pub type N999 = NInt<U999>;
pub type U1000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1000 = PInt<U1000>; pub type N1000 = NInt<U1000>;
pub type U1001 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1001 = PInt<U1001>; pub type N1001 = NInt<U1001>;
pub type U1002 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1002 = PInt<U1002>; pub type N1002 = NInt<U1002>;
pub type U1003 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1003 = PInt<U1003>; pub type N1003 = NInt<U1003>;
pub type U1004 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1004 = PInt<U1004>; pub type N1004 = NInt<U1004>;
pub type U1005 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1005 = PInt<U1005>; pub type N1005 = NInt<U1005>;
pub type U1006 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1006 = PInt<U1006>; pub type N1006 = NInt<U1006>;
pub type U1007 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1007 = PInt<U1007>; pub type N1007 = NInt<U1007>;
pub type U1008 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
```

```
pub type P1008 = PInt<U1008>; pub type N1008 = NInt<U1008>;
pub type U1009 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1009 = PInt<U1009>; pub type N1009 = NInt<U1009>;
pub type U1010 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1010 = PInt<U1010>; pub type N1010 = NInt<U1010>;
pub type U1011 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1011 = PInt<U1011>; pub type N1011 = NInt<U1011>;
pub type U1012 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1012 = PInt<U1012>; pub type N1012 = NInt<U1012>;
pub type U1013 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1013 = PInt<U1013>; pub type N1013 = NInt<U1013>;
pub type U1014 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1014 = PInt<U1014>; pub type N1014 = NInt<U1014>;
pub type U1015 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1015 = PInt<U1015>; pub type N1015 = NInt<U1015>;
pub type U1016 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1016 = PInt<U1016>; pub type N1016 = NInt<U1016>;
pub type U1017 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1017 = PInt<U1017>; pub type N1017 = NInt<U1017>;
pub type U1018 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1018 = PInt<U1018>; pub type N1018 = NInt<U1018>;
pub type U1019 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1019 = PInt<U1019>; pub type N1019 = NInt<U1019>;
pub type U1020 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1020 = PInt<U1020>; pub type N1020 = NInt<U1020>;
pub type U1021 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1021 = PInt<U1021>; pub type N1021 = NInt<U1021>;
pub type U1022 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1022 = PInt<U1022>; pub type N1022 = NInt<U1022>;
pub type U1023 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1023 = PInt<U1023>; pub type N1023 = NInt<U1023>;
pub type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1024 = PInt<U1024>; pub type N1024 = NInt<U1024>;
pub type U2048 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2048 = PInt<U2048>; pub type N2048 = NInt<U2048>;
pub type U4096 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4096 = PInt<U4096>; pub type N4096 = NInt<U4096>;
pub type U8192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P8192 = PInt<U8192>; pub type N8192 = NInt<U8192>;
pub type U16384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P16384 = PInt<U16384>; pub type N16384 = NInt<U16384>;
pub type U32768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P32768 = PInt<U32768>; pub type N32768 = NInt<U32768>;
pub type U65536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P65536 = PInt<U65536>; pub type N65536 = NInt<U65536>;
pub type U131072 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P131072 = PInt<U131072>; pub type N131072 = NInt<U131072>;
pub type U262144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P262144 = PInt<U262144>; pub type N262144 = NInt<U262144>;
pub type U524288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P524288 = PInt<U524288>; pub type N524288 = NInt<U524288>;
pub type U1048576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
```

```
pub type P1048576 = PInt<U1048576>; pub type N1048576 = NInt<U1048576>;
pub type U2097152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P2097152 = PInt<U2097152>; pub type N2097152 = NInt<U2097152>;
pub type U4194304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P4194304 = PInt<U4194304>; pub type N4194304 = NInt<U4194304>;
pub type U8388608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P8388608 = PInt<U8388608>; pub type N8388608 = NInt<U8388608>;
pub type U16777216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P16777216 = PInt<U16777216>; pub type N16777216 = NInt<U167772
pub type U33554432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P33554432 = PInt<U33554432>; pub type N33554432 = NInt<U335544
pub type U67108864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P67108864 = PInt<U67108864>; pub type N67108864 = NInt<U671088
pub type U134217728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P134217728 = PInt<U134217728>; pub type N134217728 = NInt<U134
pub type U268435456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P268435456 = PInt<U268435456>; pub type N268435456 = NInt<U268
pub type U536870912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P536870912 = PInt<U536870912>; pub type N536870912 = NInt<U536
pub type U1073741824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1073741824 = PInt<U1073741824>; pub type N1073741824 = NInt<U
pub type U2147483648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P2147483648 = PInt<U2147483648>; pub type N2147483648 = NInt<U
pub type U4294967296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P4294967296 = PInt<U4294967296>; pub type N4294967296 = NInt<U
pub type U8589934592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P8589934592 = PInt<U8589934592>; pub type N8589934592 = NInt<U
pub type U17179869184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P17179869184 = PInt<U17179869184>; pub type N17179869184 = NIn
pub type U34359738368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P34359738368 = PInt<U34359738368>; pub type N34359738368 = NIn
pub type U68719476736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P68719476736 = PInt<U68719476736>; pub type N68719476736 = NIn
pub type U137438953472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P137438953472 = PInt<U137438953472>; pub type N137438953472 =
pub type U274877906944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P274877906944 = PInt<U274877906944>; pub type N274877906944 =
pub type U549755813888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P549755813888 = PInt<U549755813888>; pub type N549755813888 =
pub type U1099511627776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1099511627776 = PInt<U1099511627776>; pub type N1099511627776
pub type U2199023255552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P2199023255552 = PInt<U2199023255552>; pub type N2199023255552
pub type U4398046511104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P4398046511104 = PInt<U4398046511104>; pub type N4398046511104
pub type U8796093022208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P8796093022208 = PInt<U8796093022208>; pub type N8796093022208
pub type U17592186044416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P17592186044416 = PInt<U17592186044416>; pub type N17592186044
pub type U35184372088832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P35184372088832 = PInt<U35184372088832>; pub type N35184372088
pub type U70368744177664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
```

```
pub type P70368744177664 = PInt<U70368744177664>; pub type N7036874417
pub type U140737488355328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P140737488355328 = PInt<U140737488355328>; pub type N140737488
pub type U281474976710656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P281474976710656 = PInt<U281474976710656>; pub type N281474976
pub type U562949953421312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P562949953421312 = PInt<U562949953421312>; pub type N562949953
pub type U1125899906842624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P1125899906842624 = PInt<U1125899906842624>; pub type N1125899
pub type U2251799813685248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P2251799813685248 = PInt<U2251799813685248>; pub type N2251799
pub type U4503599627370496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P4503599627370496 = PInt<U4503599627370496>; pub type N4503599
pub type U9007199254740992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P9007199254740992 = PInt<U9007199254740992>; pub type N9007199
pub type U18014398509481984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P18014398509481984 = PInt<U18014398509481984>; pub type N18014
pub type U36028797018963968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P36028797018963968 = PInt<U36028797018963968>; pub type N36028
pub type U72057594037927936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P72057594037927936 = PInt<U72057594037927936>; pub type N72057
pub type U144115188075855872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P144115188075855872 = PInt<U144115188075855872>; pub type N144
pub type U288230376151711744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P288230376151711744 = PInt<U288230376151711744>; pub type N288
pub type U576460752303423488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P576460752303423488 = PInt<U576460752303423488>; pub type N576
pub type U1152921504606846976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1152921504606846976 = PInt<U1152921504606846976>; pub type N1
pub type U2305843009213693952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2305843009213693952 = PInt<U2305843009213693952>; pub type N2
pub type U4611686018427387904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4611686018427387904 = PInt<U4611686018427387904>; pub type N4
pub type U9223372036854775808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type U10000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P10000 = PInt<U10000>; pub type N10000 = NInt<U10000>;
pub type U100000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P100000 = PInt<U100000>; pub type N100000 = NInt<U100000>;
pub type U1000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1000000 = PInt<U1000000>; pub type N1000000 = NInt<U1000000>;
pub type U10000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P10000000 = PInt<U10000000>; pub type N10000000 = NInt<U100000
pub type U100000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P100000000 = PInt<U100000000>; pub type N100000000 = NInt<U100
pub type U1000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1000000000 = PInt<U1000000000>; pub type N1000000000 = NInt<U
pub type U10000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P10000000000 = PInt<U10000000000>; pub type N10000000000 = NIn
pub type U100000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P100000000000 = PInt<U100000000000>; pub type N100000000000 =
pub type U1000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1000000000000 = PInt<U1000000000000>; pub type N1000000000000
```

```
    pub type U10000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P10000000000000 = PInt<U10000000000000>; pub type N10000000000
    pub type U100000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P100000000000000 = PInt<U100000000000000>; pub type N100000000
    pub type U1000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P1000000000000000 = PInt<U1000000000000000>; pub type N1000000
    pub type U10000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P10000000000000000 = PInt<U10000000000000000>; pub type N10000
    pub type U100000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P100000000000000000 = PInt<U100000000000000000>; pub type N100
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P1000000000000000000 = PInt<U1000000000000000000>; pub type N1
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
}
```

File: ./target/package/catalysh-0.0.2/target/debug/build/typenum-056

```
extern crate typenum;

use std::ops::*;
use std::cmp::Ordering;
use typenum::*;

#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitOrU0 = <<A as BitOr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitOrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_BitXor_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitXorU0 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitXorU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU0 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU0 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0AddU0 = <<A as Add<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0AddU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MaxU0 = <<A as Max<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MaxU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0GcdU0 = <<A as Gcd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0GcdU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Sub_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0SubU0 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0SubU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_0() {
    type A = UTerm;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_0() {
    type A = UTerm;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U0CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_1() {
    type A = UTerm;
```

```
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU1 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitXorU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU1 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU1 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0AddU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0MinU1 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU1 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU1 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU1 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU2 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitXorU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU2 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU2 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0AddU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU2 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_Mul_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU2 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0PowU2 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU3 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitXorU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU3 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU3 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0AddU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU3 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_3() {
    type A = UTerm;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU3 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU3 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_BitOr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU4 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitXorU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU4 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U0AddU4 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0AddU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU4 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU4 = <<A as Mul<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0MulU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU4 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_0_Cmp_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU5 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitXorU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U0ShlU5 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU5 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0AddU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU5 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU5 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_5() {
    type A = UTerm;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU5 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU0 = <<A as BitOr<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U1BitOrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU0 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitXorU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShlU0 = <<A as Shl<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShlU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShrU0 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU0 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1AddU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Min_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU0 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU0 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1SubU0 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1SubU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U1CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitXorU1 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitXorU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU1 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1ShlU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU1 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1AddU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_1() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1SubU1 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1SubU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U1MulU1 = <<A as Mul<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MulU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1DivU1 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1DivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_PartialDiv_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U1PartialDivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU1 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU2 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitXorU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U1ShlU2 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1ShlU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU2 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1AddU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU2 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U1MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU2 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MulU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Pow_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU2 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U1BitXorU3 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1BitXorU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU3 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U1ShlU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU3 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1AddU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU3 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_Max_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU3 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MulU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_3() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU3 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U1BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU4 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitXorU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU4 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U1ShlU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU4 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1AddU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_1_Min_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU4 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU4 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MulU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU4 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU5 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1BitXorU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U1ShlU5 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U1ShlU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_5() {
    type A = UInt<UTerm, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU5 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U1AddU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU5 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U1MulU5 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MulU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU5 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU5 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU0 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU0 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitXorU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU0 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShlU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2ShrU0 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU0 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2AddU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU0 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU0 = <<A as Gcd<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2GcdU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2SubU0 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2SubU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U2CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU1 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitXorU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU1 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2ShlU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU1 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2AddU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU1 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2SubU1 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2SubU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU1 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MulU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2DivU1 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2DivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U2PartialDivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU1 = <<A as Pow<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2PowU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitXorU2 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitXorU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU2 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2ShlU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU2 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2AddU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2SubU2 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2SubU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2MulU2 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MulU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U2PartialDivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU2 = <<A as Pow<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U2PowU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU3 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU3 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2BitXorU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU3 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2ShlU3 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU3 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2AddU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU3 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU3 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2MulU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU3 = <<A as Pow<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2PowU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU4 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitOrU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2BitXorU4 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitXorU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU4 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2ShlU4 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU4 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2AddU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU4 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2MinU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU4 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU4 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2MulU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_2_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU4 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU4 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2PowU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU5 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitXorU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU5 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U2ShlU5 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU5 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2AddU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_2_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU5 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU5 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U2MulU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU5 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2PowU5 = <<A as Pow<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2PowU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU0 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU0 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitXorU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShlU0 = <<A as Shl<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShlU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU0 = <<A as Shr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU0 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3AddU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU0 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU0 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3SubU0 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3SubU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U3CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU1 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitXorU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU1 = <<A as Shl<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3ShlU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU1 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3AddU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU1 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3SubU1 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3SubU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU1 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MulU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU1 = <<A as Div<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3DivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U3>>::Outpu

    assert_eq!(<U3PartialDivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU1 = <<A as Pow<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3PowU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3BitXorU2 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitXorU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU2 = <<A as Shl<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3ShlU2 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU2 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3AddU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U3MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU2 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU2 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3SubU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU2 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3MulU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU2 = <<A as Pow<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3PowU2 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U3BitAndU3 = <<A as BitAnd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitAndU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitXorU3 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitXorU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU3 = <<A as Shl<B>>::Output as Same<U24>>::Output;

    assert_eq!(<U3ShlU3 as Unsigned>::to_u64(), <U24 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU3 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3AddU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3SubU3 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3SubU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU3 = <<A as Mul<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3MulU3 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U3PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U3PartialDivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU3 = <<A as Pow<B>>::Output as Same<U27>>::Output;

    assert_eq!(<U3PowU3 as Unsigned>::to_u64(), <U27 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU4 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_3_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU4 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitXorU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U48 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU4 = <<A as Shl<B>>::Output as Same<U48>>::Output;

    assert_eq!(<U3ShlU4 as Unsigned>::to_u64(), <U48 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU4 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3AddU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3MinU4 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU4 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3MulU4 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U3DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU4 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U3PowU4 = <<A as Pow<B>>::Output as Same<U81>>::Output;

    assert_eq!(<U3PowU4 as Unsigned>::to_u64(), <U81 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU5 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3BitXorU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU5 = <<A as Shl<B>>::Output as Same<U96>>::Output;

    assert_eq!(<U3ShlU5 as Unsigned>::to_u64(), <U96 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U3AddU5 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U3AddU5 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU5 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU5 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U3MulU5 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_3_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU5 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U3PowU5 = <<A as Pow<B>>::Output as Same<U243>>::Output;

    assert_eq!(<U3PowU5 as Unsigned>::to_u64(), <U243 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU0 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU0 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitXorU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU0 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShlU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU0 = <<A as Shr<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4ShrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU0 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4AddU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU0 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU0 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU0 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4SubU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U4CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
```

```
    type U4BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU1 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitXorU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU1 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4ShlU1 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU1 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4AddU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU1 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU1 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4SubU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU1 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MulU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU1 = <<A as Div<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4DivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U4PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U4>>::Outpu

    assert_eq!(<U4PartialDivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU1 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4PowU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU2 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitOrU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU2 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitXorU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU2 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4ShlU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU2 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4AddU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU2 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU2 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4SubU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU2 = <<A as Mul<B>>::Output as Same<U8>>::Output;
```

```
    assert_eq!(<U4MulU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U4PartialDivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU2 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4PowU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU3 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitXorU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
```

```
    type U4ShlU3 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U4ShlU3 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU3 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4AddU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU3 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU3 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4SubU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU3 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U4MulU3 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4PowU3 = <<A as Pow<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4PowU3 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitXorU4 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitXorU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU4 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4ShlU4 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU4 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4AddU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_4_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4SubU4 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4SubU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MulU4 = <<A as Mul<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4MulU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U4PartialDivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0

    #[allow(non_camel_case_types)]
    type U4PowU4 = <<A as Pow<B>>::Output as Same<U256>>::Output;
```

```
    assert_eq!(<U4PowU4 as Unsigned>::to_u64(), <U256 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU5 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU5 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4BitXorU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0

    #[allow(non_camel_case_types)]
    type U4ShlU5 = <<A as Shl<B>>::Output as Same<U128>>::Output;

    assert_eq!(<U4ShlU5 as Unsigned>::to_u64(), <U128 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU5 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U4AddU5 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU5 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U4MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU5 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U4MulU5 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4RemU5 = <<A as Rem<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4RemU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTe

    #[allow(non_camel_case_types)]
    type U4PowU5 = <<A as Pow<B>>::Output as Same<U1024>>::Output;

    assert_eq!(<U4PowU5 as Unsigned>::to_u64(), <U1024 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU0 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitXorU0 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitXorU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShlU0 = <<A as Shl<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShlU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU0 = <<A as Shr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU0 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5AddU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U5MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU0 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU0 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU0 = <<A as Sub<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5SubU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U5CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
```

```
    type U5BitXorU1 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitXorU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU1 = <<A as Shl<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5ShlU1 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU1 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5AddU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU1 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU1 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5SubU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU1 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MulU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU1 = <<A as Div<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5DivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U5>>::Outpu

    assert_eq!(<U5PartialDivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU1 = <<A as Pow<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5PowU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU1 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<U5CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU2 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU2 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitXorU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU2 = <<A as Shl<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5ShlU2 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU2 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5AddU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU2 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU2 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5SubU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU2 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5MulU2 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U5RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU2 = <<A as Pow<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5PowU2 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU3 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5BitXorU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU3 = <<A as Shl<B>>::Output as Same<U40>>::Output;

    assert_eq!(<U5ShlU3 as Unsigned>::to_u64(), <U40 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU3 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U5AddU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U5MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU3 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU3 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5SubU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU3 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U5MulU3 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U5PowU3 = <<A as Pow<B>>::Output as Same<U125>>::Output;

    assert_eq!(<U5PowU3 as Unsigned>::to_u64(), <U125 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU4 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitXorU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU4 = <<A as Shl<B>>::Output as Same<U80>>::Output;

    assert_eq!(<U5ShlU4 as Unsigned>::to_u64(), <U80 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U5ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU4 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U5AddU4 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU4 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_5_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU4 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5SubU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU4 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5MulU4 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1
```

```
    #[allow(non_camel_case_types)]
    type U5PowU4 = <<A as Pow<B>>::Output as Same<U625>>::Output;

    assert_eq!(<U5PowU4 as Unsigned>::to_u64(), <U625 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU5 = <<A as BitAnd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitAndU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitXorU5 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitXorU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1

    #[allow(non_camel_case_types)]
    type U5ShlU5 = <<A as Shl<B>>::Output as Same<U160>>::Output;

    assert_eq!(<U5ShlU5 as Unsigned>::to_u64(), <U160 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU5 = <<A as Add<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5AddU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU5 = <<A as Min<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MinU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5SubU5 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5SubU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U25 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU5 = <<A as Mul<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5MulU5 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U5DivU5 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U5PartialDivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type U5PowU5 = <<A as Pow<B>>::Output as Same<U3125>>::Output;

    assert_eq!(<U5PowU5 as Unsigned>::to_u64(), <U3125 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN5 = <<A as Add<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5AddN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5SubN5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5SubN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulN5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5MulN5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MaxN5 = <<A as Max<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MaxN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu
```

```
    assert_eq!(<N5PartialDivN5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN4 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5AddN4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5SubN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulN4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N5MulN4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN3 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5AddN3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5SubN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N5MulN3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N5RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN2 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5AddN2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5SubN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulN2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N5MulN2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN1 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5AddN1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5SubN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN1 = <<A as Mul<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<N5MulN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5DivN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<N5PartialDivN1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Add_0 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Add_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5Sub_0 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Sub_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Min_0 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Min_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N5Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5AddP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP1 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5SubP1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MulP1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MulP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP1 = <<A as Div<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<N5DivP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<N5PartialDivP1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PowP1 = <<A as Pow<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5PowP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5AddP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP2 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5SubP2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulP2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5MulP2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N5MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5AddP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP3 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5SubP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N5MulP3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MinP3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N5MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N5RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type N5PowP3 = <<A as Pow<B>>::Output as Same<N125>>::Output;

    assert_eq!(<N5PowP3 as Integer>::to_i64(), <N125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5AddP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP4 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5SubP4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulP4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N5MulP4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N5MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N5DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type N5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<N5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5AddP5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5AddP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N5_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP5 = <<A as Sub<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5SubP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulP5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<N5MulP5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N5PartialDivP5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N5PowP5 = <<A as Pow<B>>::Output as Same<N3125>>::Output;
```

```
    assert_eq!(<N5PowP5 as Integer>::to_i64(), <N3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN5 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4AddN5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN5 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4SubN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N4MulN5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN5 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N4RemN5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN4 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4AddN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4SubN4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4SubN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4MulN4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N4_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N4PartialDivN4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN3 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4AddN3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4SubN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N4MulN3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN2 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4AddN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubN2 = <<A as Sub<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N4SubN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N4MulN2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N4_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N4PartialDivN2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4AddN1 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4AddN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4SubN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MulN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4DivN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<N4PartialDivN1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Add_0 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Add_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Sub_0 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Sub_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Min_0 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N4Min_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N4Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4AddP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP1 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4SubP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MulP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4DivP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<N4PartialDivP1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PowP1 = <<A as Pow<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4PowP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddP2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4AddP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP2 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4SubP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MulP2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4MulP2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N4DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N4PartialDivP2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4AddP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP3 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4SubP3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N4MulP3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N4MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N64 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP3 = <<A as Pow<B>>::Output as Same<N64>>::Output;

    assert_eq!(<N4PowP3 as Integer>::to_i64(), <N64 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4AddP4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4AddP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP4 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4SubP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<N4MulP4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MinP4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N4PartialDivP4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type N4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<N4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP5 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4AddP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP5 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4SubP5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N4MulP5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP5 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemP5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1024 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N4PowP5 = <<A as Pow<B>>::Output as Same<N1024>>::Output;

    assert_eq!(<N4PowP5 as Integer>::to_i64(), <N1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN5 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3AddN5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN5 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3SubN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N3MulN5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxN5 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN4 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3AddN4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3SubN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N3MulN4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN4 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN4 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<N3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN3 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3AddN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3SubN3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3SubN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3MulN3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N3PartialDivN3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN2 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3AddN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3SubN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N3MulN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3AddN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3SubN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN1 = <<A as Mul<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<N3MulN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3DivN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<N3PartialDivN1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Add_0 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Add_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N3Sub_0 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Sub_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Min_0 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Min_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N3Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3AddP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3SubP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MulP1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MulP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP1 = <<A as Div<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3DivP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<N3PartialDivP1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP1 = <<A as Pow<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3PowP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3AddP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP2 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3SubP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3MulP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N3MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3AddP3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3AddP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP3 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3SubP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N3MulP3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinP3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N3PartialDivP3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N27 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP3 = <<A as Pow<B>>::Output as Same<N27>>::Output;

    assert_eq!(<N3PowP3 as Integer>::to_i64(), <N27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3AddP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP4 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3SubP4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N3MulP4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP4 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<N3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N3_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP5 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3AddP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP5 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3SubP5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N3MulP5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP5 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP5 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N243 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type N3PowP5 = <<A as Pow<B>>::Output as Same<N243>>::Output;
```

```
    assert_eq!(<N3PowP5 as Integer>::to_i64(), <N243 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN5 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2AddN5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN5 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2SubN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N2MulN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN5 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N2RemN5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN4 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2AddN4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubN4 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2SubN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N2MulN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN4 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2RemN4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN3 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2AddN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2SubN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N2MulN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN3 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN2 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2AddN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2SubN2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2SubN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN2 = <<A as Mul<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<N2MulN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N2PartialDivN2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2AddN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2SubN1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2SubN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MulN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivN1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2DivN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N2PartialDivN1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2Add_0 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Add_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Sub_0 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Sub_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Min_0 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Min_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N2Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2AddP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2SubP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MulP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivP1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2DivP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N2PartialDivP1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP1 = <<A as Pow<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2PowP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2AddP2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2AddP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP2 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2SubP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MulP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MinP2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N2PartialDivP2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2AddP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP3 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2SubP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2MulP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP3 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type N2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP3 = <<A as Pow<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2PowP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddP4 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2AddP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP4 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2SubP4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2MulP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP4 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;
```

```
    assert_eq!(<N2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP5 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2AddP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP5 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2SubP5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N2MulP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP5 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N2RemP5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N32 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP5 = <<A as Pow<B>>::Output as Same<N32>>::Output;

    assert_eq!(<N2PowP5 as Integer>::to_i64(), <N32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN5 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1AddN5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN5 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1SubN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Mul_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MulN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN5 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN4 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1AddN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN4 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1SubN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MulN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN4 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N4() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN4 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<N1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN3 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1AddN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1SubN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MulN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Max_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN3 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1PowN3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1AddN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1SubN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MulN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN2 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N2() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1AddN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1SubN1 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N1SubN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MulN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Div_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivN1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1DivN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N1PartialDivN1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Add_0 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Add_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Sub_0 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Sub_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Min_0 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Min_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Max__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N1Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N1AddP1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1AddP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1SubP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MulP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivP1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1DivP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N1PartialDivP1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N1_Pow_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1AddP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1SubP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N1MulP2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MulP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP2 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1AddP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N1SubP3 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1SubP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MulP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP3 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P4() {
    type A = NInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP4 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1AddP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP4 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1SubP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MulP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP4 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type N1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP5 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1AddP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP5 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1SubP5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MulP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1MinP5 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP5 = <<A as Rem<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N1RemP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN5 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0AddN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN5 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0SubN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N5() {
    type A = Z0;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN5 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type _0DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN4 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0AddN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Sub_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN4 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0SubN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN4 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type _0GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN3 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0AddN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN3 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0SubN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N3() {
    type A = Z0;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN3 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type _0PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0AddN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0SubN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Min_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN2 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0AddN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0SubN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN1 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N1() {
    type A = Z0;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Add_0 = <<A as Add<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0Add_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Sub_0 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Sub_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_Gcd__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Gcd_0 = <<A as Gcd<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Gcd_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow__0() {
    type A = Z0;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp__0() {
    type A = Z0;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type _0Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<_0Cmp_0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0AddP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0SubP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP1 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP1 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P1() {
    type A = Z0;
```

```
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0AddP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0SubP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP2 = <<A as Min<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MinP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_PartialDiv_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP2 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP3 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0AddP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP3 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0SubP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP3 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP3 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P3() {
    type A = Z0;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP4 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0AddP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP4 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0SubP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP4 = <<A as Min<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MinP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test__0_PartialDiv_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP4 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddP5 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0AddP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0SubP5 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0SubP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP5 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP5 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P5() {
    type A = Z0;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN5 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1AddN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN5 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1SubN5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MulN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<P1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN5 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Pow_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN4 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1AddN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN4 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1SubN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P1MulN4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MulN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN4 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1AddN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P1SubN3 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1SubN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MulN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN3 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N2() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1AddN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1SubN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MulN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1MaxN2 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1AddN1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1AddN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1SubN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MulN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivN1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1DivN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P1PartialDivN1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Add_0 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Add_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub__0() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Sub_0 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Sub_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Max_0 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Max_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P1Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1AddP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1SubP1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1SubP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Mul_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MulP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1DivP1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1DivP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P1PartialDivP1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP1 as Ord>::to_ordering(), Ordering::Equal);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1AddP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1SubP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MulP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP2 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P2() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP3 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1AddP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1SubP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MulP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Min_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP3 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP4 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1AddP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP4 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1SubP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MulP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP4 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P4() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP5 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P1AddP5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP5 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1SubP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MulP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP5 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P1_Gcd_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN5 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2AddN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN5 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2SubN5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P2MulN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN5 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddN4 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2AddN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN4 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2SubN4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P2MulN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<P2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN4 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2AddN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN3 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2SubN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P2MulN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN3 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2AddN2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2AddN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN2 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2SubN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MulN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P2PartialDivN2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2AddN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2SubN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MulN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P2DivN1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2DivN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P2PartialDivN1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Add_0 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Add_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Sub_0 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Sub_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Max_0 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Max_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P2Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2AddP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2SubP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MulP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivP1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2DivP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P2PartialDivP1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP1 = <<A as Pow<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2PowP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP1 = <A as Cmp<B>>::Output;
```

```
    assert_eq!(<P2CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP2 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2AddP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2SubP2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2SubP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MulP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P2PartialDivP2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP3 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2AddP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2SubP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2MulP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP3 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP3 = <<A as Pow<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2PowP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP4 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P2AddP4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubP4 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2SubP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2MulP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP4 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P2_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP5 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2AddP5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP5 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2SubP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P2MulP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP5 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P32 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP5 = <<A as Pow<B>>::Output as Same<P32>>::Output;

    assert_eq!(<P2PowP5 as Integer>::to_i64(), <P32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN5 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3AddN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN5 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3SubN5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN5 = <<A as Mul<B>>::Output as Same<N15>>::Output;
```

```
    assert_eq!(<P3MulN5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN5 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3AddN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN4 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3SubN4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P3MulN4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P3MulN4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN4 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3AddN3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3AddN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN3 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3SubN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3MulN3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<P3MulN3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P3DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P3PartialDivN3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3AddN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN2 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3SubN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P3MulN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3AddN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3SubN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MulN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3DivN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<P3PartialDivN1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Add_0 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Add_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Sub_0 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Sub_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Max_0 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Max_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P3Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP1 = <<A as Add<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P3AddP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3SubP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MulP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3DivP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<P3PartialDivP1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3PowP1 = <<A as Pow<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3PowP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP2 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3AddP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3SubP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3MulP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP3 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3AddP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3SubP3 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P3SubP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3MulP3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P3PartialDivP3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P27 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP3 = <<A as Pow<B>>::Output as Same<P27>>::Output;

    assert_eq!(<P3PowP3 as Integer>::to_i64(), <P27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP4 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3AddP4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3SubP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P3MulP4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP4 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<P3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP5 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3AddP5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP5 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3SubP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP5 = <<A as Mul<B>>::Output as Same<P15>>::Output;
```

```
    assert_eq!(<P3MulP5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP5 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P3_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P243 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type P3PowP5 = <<A as Pow<B>>::Output as Same<P243>>::Output;

    assert_eq!(<P3PowP5 as Integer>::to_i64(), <P243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN5 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4AddN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P4SubN5 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4SubN5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P4MulN5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN5 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemN5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4AddN4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4AddN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4SubN4 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4SubN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<P4MulN4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P4PartialDivN4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4AddN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN3 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4SubN3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P4MulN3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P4MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P4MaxN3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddN2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4AddN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN2 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4SubN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P4MulN2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4MaxN2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu
```

```
    assert_eq!(<P4PartialDivN2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4AddN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN1 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4SubN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MulN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4DivN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
```

```
    type P4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<P4PartialDivN1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Add_0 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Add_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Sub_0 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Sub_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Max_0 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Max_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P4Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP1 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4AddP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4SubP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MulP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4DivP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<P4PartialDivP1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PowP1 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4PowP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP2 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P4AddP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubP2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4SubP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4MulP2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P4PartialDivP2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>
```

```
    #[allow(non_camel_case_types)]
    type P4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP3 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4AddP3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4SubP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P4MulP3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P64 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP3 = <<A as Pow<B>>::Output as Same<P64>>::Output;

    assert_eq!(<P4PowP3 as Integer>::to_i64(), <P64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP4 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4AddP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4SubP4 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P4SubP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4MulP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P4_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P4PartialDivP4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type P4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<P4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP5 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4AddP5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP5 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4SubP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P4MulP5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP5 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemP5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1024 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P4PowP5 = <<A as Pow<B>>::Output as Same<P1024>>::Output;

    assert_eq!(<P4PowP5 as Integer>::to_i64(), <P1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5AddN5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5AddN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN5 = <<A as Sub<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5SubN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulN5 = <<A as Mul<B>>::Output as Same<N25>>::Output;
```

```
    assert_eq!(<P5MulN5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P5PartialDivN5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5AddN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5SubN4 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5SubN4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulN4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P5MulN4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P5MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5AddN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P5SubN3 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5SubN3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P5MulN3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P5MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5AddN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN2 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5SubN2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulN2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P5MulN2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5AddN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN1 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5SubN1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MulN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5DivN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<P5PartialDivN1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Add_0 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Add_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Sub_0 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Sub_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Max_0 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Max_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P5Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP1 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P5AddP1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5SubP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MulP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5DivP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<P5PartialDivP1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5PowP1 = <<A as Pow<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5PowP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP2 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5AddP2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5SubP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulP2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5MulP2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP3 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5AddP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP3 = <<A as Sub<B>>::Output as Same<P2>>::Output;
```

```
    assert_eq!(<P5SubP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P5MulP3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1
```

```
    #[allow(non_camel_case_types)]
    type P5PowP3 = <<A as Pow<B>>::Output as Same<P125>>::Output;

    assert_eq!(<P5PowP3 as Integer>::to_i64(), <P125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P5AddP4 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5AddP4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5SubP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulP4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P5MulP4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type P5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<P5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP5 = <<A as Add<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5AddP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5SubP5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5SubP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulP5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5MulP5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP5 = <<A as Min<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P5MinP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_P5_PartialDiv_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P5PartialDivP5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P5PowP5 = <<A as Pow<B>>::Output as Same<P3125>>::Output;

    assert_eq!(<P5PowP5 as Integer>::to_i64(), <P3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN5 = <<A as Neg>::Output as Same<P5>>::Output;
    assert_eq!(<NegN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

```
#[test]
#[allow(non_snake_case)]
fn test_N4_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN4 = <<A as Neg>::Output as Same<P4>>::Output;
    assert_eq!(<NegN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN3 = <<A as Neg>::Output as Same<P3>>::Output;
    assert_eq!(<NegN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN2 = <<A as Neg>::Output as Same<P2>>::Output;
    assert_eq!(<NegN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
```

```
fn test_N2_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Neg() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegN1 = <<A as Neg>::Output as Same<P1>>::Output;
    assert_eq!(<NegN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Abs() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Neg() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Neg_0 = <<A as Neg>::Output as Same<_0>>::Output;
    assert_eq!(<Neg_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Abs() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Abs_0 = <<A as Abs>::Output as Same<_0>>::Output;
    assert_eq!(<Abs_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Neg() {
    type A = PInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegP1 = <<A as Neg>::Output as Same<N1>>::Output;
    assert_eq!(<NegP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Abs() {
    type A = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP2 = <<A as Neg>::Output as Same<N2>>::Output;
    assert_eq!(<NegP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP3 = <<A as Neg>::Output as Same<N3>>::Output;
    assert_eq!(<NegP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type AbsP3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP4 = <<A as Neg>::Output as Same<N4>>::Output;
    assert_eq!(<NegP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP5 = <<A as Neg>::Output as Same<N5>>::Output;
    assert_eq!(<NegP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

File: ./target/package/catalysh-0.0.2/target/debug/build/typenum-056

```
/**
Convenient type operations.
```

Any types representing values must be able to be expressed as `ident`s. Tha
in scope.

For example, `P5` is okay, but `typenum::P5` is not.

You may combine operators arbitrarily, although doing so excessively may re
recursion limit.

Example
```rust
#![recursion_limit="128"]
#[macro_use] extern crate typenum;
use typenum::consts::*;

fn main() {
    assert_type!(
        op!(min((P1 - P2) * (N3 + N7), P5 * (P3 + P4)) == P10)
    );
}
```
Operators are evaluated based on the operator precedence outlined
[here](https://doc.rust-lang.org/reference.html#operator-precedence).

The full list of supported operators and functions is as follows:

`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, `==`, `!=`, `<=`, `>=`,

They all expand to type aliases defined in the `operator_aliases` module. H
including examples:

Operator `*`. Expands to `Prod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 * P3), P6);
# }
```

Operator `/`. Expands to `Quot`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 / P2), P3);
# }
```

Operator `%`. Expands to `Mod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 % P3), P2);
# }
```

Operator `+`. Expands to `Sum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 + P3), P5);
# }
```

Operator `-`. Expands to `Diff`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 - P3), N1);
# }
```

Operator `<<`. Expands to `Shleft`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U1 << U5), U32);
# }
```

Operator `>>`. Expands to `Shright`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
```

```
assert_type_eq!(op!(U32 >> U5), U1);
# }
```

Operator `&`. Expands to `And`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 & U3), U1);
# }
```

Operator `^`. Expands to `Xor`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 ^ U3), U6);
# }
```

Operator `|`. Expands to `Or`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 | U3), U7);
# }
```

Operator `==`. Expands to `Eq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 == P3 + P2), True);
# }
```

Operator `!=`. Expands to `NotEq`.

```rust
```

```
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 != P3 + P2), False);
# }
```

Operator `<=`. Expands to `LeEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 <= P3 + P2), False);
# }
```

Operator `>=`. Expands to `GrEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 >= P3 + P2), True);
# }
```

Operator `<`. Expands to `Le`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P4 < P3 + P2), True);
# }
```

Operator `>`. Expands to `Gr`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 < P3 + P2), False);
# }
```

Operator `cmp`. Expands to `Compare`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cmp(P2, P3)), Less);
# }
```

Operator `sqr`. Expands to `Square`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqr(P2)), P4);
# }
```

Operator `sqrt`. Expands to `Sqrt`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqrt(U9)), U3);
# }
```

Operator `abs`. Expands to `AbsVal`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(abs(N2)), P2);
# }
```

Operator `cube`. Expands to `Cube`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cube(P2)), P8);
# }
```

```

---
Operator `pow`. Expands to `Exp`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(pow(P2, P3)), P8);
# }
```

Operator `min`. Expands to `Minimum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(min(P2, P3)), P2);
# }
```

Operator `max`. Expands to `Maximum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(max(P2, P3)), P3);
# }
```

Operator `log2`. Expands to `Log2`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(log2(U9)), U3);
# }
```

Operator `gcd`. Expands to `Gcf`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
```
```

```
# fn main() {
assert_type_eq!(op!(gcd(U9, U21)), U3);
# }
```

*/
#[macro_export(local_inner_macros)]
macro_rules! op {
    ($($tail:tt)*) => (__op_internal__!($($tail)*));
}

    #[doc(hidden)]
    #[macro_export(local_inner_macros)]
    macro_rules! __op_internal__ {

(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cmp $($tail:tt)
    __op_internal__!(@stack[Compare, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqr $($tail:tt)
    __op_internal__!(@stack[Square, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqrt $($tail:tt
    __op_internal__!(@stack[Sqrt, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: abs $($tail:tt)
    __op_internal__!(@stack[AbsVal, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cube $($tail:tt
    __op_internal__!(@stack[Cube, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: pow $($tail:tt)
    __op_internal__!(@stack[Exp, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: min $($tail:tt)
    __op_internal__!(@stack[Minimum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: max $($tail:tt)
    __op_internal__!(@stack[Maximum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: log2 $($tail:tt
    __op_internal__!(@stack[Log2, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: gcd $($tail:tt)
    __op_internal__!(@stack[Gcf, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: , $($ta
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail

```
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: *
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: *
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: *
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:tt)*)
    __op_internal__!(@stack[Prod, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: /
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: /
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: /
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:tt)*)
    __op_internal__!(@stack[Quot, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: %
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: %
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: %
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:tt)*)
    __op_internal__!(@stack[Mod, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: +
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: +
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: +
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: +
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: +
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:tt)*)
    __op_internal__!(@stack[Sum, $($stack,)*] @queue[$($queue,)*] @tail: $(
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: -
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: -
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: -
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: -
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: -
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:tt)*)
    __op_internal__!(@stack[Diff, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <<
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <<
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail:tt)*
    __op_internal__!(@stack[Shleft, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >>
);
```

```
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >>
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail:tt)*
    __op_internal__!(@stack[Shright, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: &
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: &
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: &
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: &
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: &
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: &
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:tt)*)
    __op_internal__!(@stack[And, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: ^
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: ^
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ^
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ^
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: ^
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ^
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ^
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:tt)*)
    __op_internal__!(@stack[Xor, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: |
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: |
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: |
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: |
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: |
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: |
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: |
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: | $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:tt)*)
    __op_internal__!(@stack[Or, $($stack,)*] @queue[$($queue,)*] @tail: $($
```

```
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: =
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: =
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ==
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ==
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: =
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ==
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ==
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: ==
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: ==
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: =
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: =
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: ==
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: ==
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:tt)*
    __op_internal__!(@stack[Eq, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
```

```
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: !
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: !
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: !=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: !=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: !
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: !=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: !=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: !=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: !=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: !
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: !
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: !=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: !=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:tt)*
    __op_internal__!(@stack[NotEq, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: <=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: <=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: <=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: <=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:tt)*
    __op_internal__!(@stack[LeEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
```

```
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: >=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: >=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: >=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: >=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:tt)*
    __op_internal__!(@stack[GrEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
```

```
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: < $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: < $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: < $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: < $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:tt)*)
    __op_internal__!(@stack[Le, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: > $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: > $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: > $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: > $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:tt)*)
    __op_internal__!(@stack[Gr, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ( $($stuff:tt)*
 => (
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*]
                     @tail: $($stuff)* RParen $($tail)*)
);
```

```
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: RParen
    __op_internal__!(@rp3 @stack[$($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
 => (
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@rp3 @stack[Compare, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Compare, $($queue,)*] @tail
);
(@rp3 @stack[Square, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[Square, $($queue,)*] @tail:
);
(@rp3 @stack[Sqrt, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Sqrt, $($queue,)*] @tail: $
);
(@rp3 @stack[AbsVal, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[AbsVal, $($queue,)*] @tail:
);
(@rp3 @stack[Cube, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Cube, $($queue,)*] @tail: $
);
(@rp3 @stack[Exp, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Exp, $($queue,)*] @tail: $(
);
(@rp3 @stack[Minimum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Minimum, $($queue,)*] @tail
);
(@rp3 @stack[Maximum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Maximum, $($queue,)*] @tail
);
(@rp3 @stack[Log2, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Log2, $($queue,)*] @tail: $
);
(@rp3 @stack[Gcf, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Gcf, $($queue,)*] @tail: $(
);
(@rp3 @stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($tail:tt
    __op_internal__!(@stack[$($stack,)*] @queue[$($queue,)*] @tail: $($tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $num:ident $($t
    __op_internal__!(@stack[$($stack,)*] @queue[$num, $($queue,)*] @tail: $
);
(@stack[] @queue[$($queue:ident,)*] @tail: ) => (
    __op_internal__!(@reverse[] @input: $($queue,)*)
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@reverse[$($revved:ident,)*] @input: $head:ident, $($tail:ident,)* ) => (
    __op_internal__!(@reverse[$head, $($revved,)*] @input: $($tail,)*)
);
```

```
(@reverse[$($revved:ident,)*] @input: ) => (
    __op_internal__!(@eval @stack[] @input[$($revved,)*])
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Prod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Prod<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Quot, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Quot<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Mod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Mod<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Sum, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Sum<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Diff, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Diff<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shleft, $($tail:ident,)*
    __op_internal__!(@eval @stack[$crate::Shleft<$b, $a>, $($stack,)*] @inp
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shright, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Shright<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[And, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::And<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Xor, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Xor<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Or, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Or<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Eq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Eq<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[NotEq, $($tail:ident,)*]
    __op_internal__!(@eval @stack[$crate::NotEq<$b, $a>, $($stack,)*] @inpu
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[LeEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::LeEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[GrEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::GrEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Le, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Le<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gr, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Gr<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Compare, $($tail:ident,)
```

```
    __op_internal__!(@eval @stack[$crate::Compare<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Exp, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Exp<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Minimum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Minimum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Maximum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Maximum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gcf, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Gcf<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Square, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Square<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Sqrt, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Sqrt<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[AbsVal, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::AbsVal<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Cube, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Cube<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Log2, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Log2<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$($stack:ty,)*] @input[$head:ident, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$head, $($stack,)*] @input[$($tail,)*])
);
(@eval @stack[$stack:ty,] @input[]) => (
    $stack
);
($($tail:tt)* ) => (
    __op_internal__!(@stack[] @queue[] @tail: $($tail)*)
);
}
```

File: ./target/package/catalysh-0.0.2/target/debug/build/libsqlite3-sys

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
```

```
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
```

```
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
```

```
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
```

```
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
```

```
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
```

```
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
```

```
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
```

```
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
```

```
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
```

```
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
```

```
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xWrite: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *const ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
    >,
    pub xSync: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFileSize: ::std::option::Option<
```

```
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xDeviceCharacteristics: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xShmMap: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        iPg: ::std::os::raw::c_int,
        pgsz: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xShmLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        offset: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
```

```
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
    pub pNext: *mut sqlite3_vfs,
    pub zName: *const ::std::os::raw::c_char,
    pub pAppData: *mut ::std::os::raw::c_void,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: sqlite3_filename,
            arg2: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
            pOutFlags: *mut ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int,
>,
pub xDelete: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        syncDir: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xAccess: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        flags: ::std::os::raw::c_int,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFullPathname: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        nOut: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xDlOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zFilename: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
    >,
    pub xRandomness: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            nByte: ::std::os::raw::c_int,
            zOut: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSleep: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            microseconds: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCurrentTime: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
    >,
    pub xGetLastError: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: ::std::os::raw::c_int,
            arg3: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCurrentTimeInt64: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            arg2: sqlite3_syscall_ptr,
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
```

```
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
```

```
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
```

```
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
```

```
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
```

```
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
```

```
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
```

```
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
```

```
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
```

```
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
```

```
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
```

```
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
```

```
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
```

```
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
```

```
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
```

```
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
```

```
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
```

```
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
```

```
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
```

```
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
```

```
    ) -> ::std::os::raw::c_int,
>,
pub xConnect: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3,
        pAux: *mut ::std::os::raw::c_void,
        argc: ::std::os::raw::c_int,
        argv: *const *const ::std::os::raw::c_char,
        ppVTab: *mut *mut sqlite3_vtab,
        arg2: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xBestIndex: ::std::option::Option<
    unsafe extern "C" fn(
        pVTab: *mut sqlite3_vtab,
        arg1: *mut sqlite3_index_info,
    ) -> ::std::os::raw::c_int,
>,
pub xDisconnect: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xDestroy: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xOpen: ::std::option::Option<
    unsafe extern "C" fn(
        pVTab: *mut sqlite3_vtab,
        ppCursor: *mut *mut sqlite3_vtab_cursor,
    ) -> ::std::os::raw::c_int,
>,
pub xClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xFilter: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        idxNum: ::std::os::raw::c_int,
        idxStr: *const ::std::os::raw::c_char,
        argc: ::std::os::raw::c_int,
        argv: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int,
>,
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
```

```
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        nArg: ::std::os::raw::c_int,
        zName: *const ::std::os::raw::c_char,
        pxFunc: *mut ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        ppArg: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xRename: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        zNew: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSavepoint: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
```

```
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
```

```
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
```

```
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
```

```
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
```

```
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
```

```
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
```

```
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
```

```
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
```

```
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
```

```
pub xUserData: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
>,
pub xColumnCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xRowCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnRow: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnTotalSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xTokenize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pText: *const ::std::os::raw::c_char,
        nText: ::std::os::raw::c_int,
        pCtx: *mut ::std::os::raw::c_void,
        xToken: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
                arg5: ::std::os::raw::c_int,
                arg6: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xPhraseCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xPhraseSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
```

```
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pAux: *mut ::std::os::raw::c_void,
        xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int,
>,
pub xGetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        bClear: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void,
```

```
    >,
    pub xPhraseFirst: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
            arg4: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNext: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
            piOff: *mut ::std::os::raw::c_int,
        ),
    >,
    pub xPhraseFirstColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNextColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
```

```
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./target/package/catalysh-0.0.2/src/app/config.rs

```
use anyhow::Result;
use serde::{Deserialize, Serialize};
use std::fs;
use std::io::{self, Write};
use std::path::PathBuf;

use dirs::config_dir;

#[derive(Serialize, Deserialize, Clone)]
pub struct Config {
    pub dnac_url: String,
    pub username: String,
    pub verify_ssl: bool,
}

impl Config {
    pub fn new(dnac_url: String, username: String, verify_ssl: bool) -> Sel
        Self {
            dnac_url,
            username,
            verify_ssl,
        }
    }
}

pub fn get_config_path() -> PathBuf {
    let mut config_path = config_dir().unwrap();
    config_path.push("catalysh");
    fs::create_dir_all(&config_path).unwrap();
    config_path.push("config.yml");
    config_path
}

pub fn get_credentials_db_path() -> PathBuf {
    let mut db_path = config_dir().unwrap();
    db_path.push("catalysh");
    db_path.push("credentials.db");
    db_path
}

/// Load configuration and trigger setup if necessary
pub fn load_config() -> Result<Config> {
    let config_path = get_config_path();
    if config_path.exists() {
        let contents = fs::read_to_string(config_path)?;
        let config: Config = serde_yaml::from_str(&contents)?;
        Ok(config)
    } else {
        println!("Configuration file not found. Starting setup...");
        let config = setup_config()?;
```

```
        save_config(&config)?;
        Ok(config)
    }
}

/// Reset the configuration and credentials
pub fn reset_config() -> Result<()> {
    let config_path = get_config_path();
    let credentials_db_path = get_credentials_db_path();

    if config_path.exists() {
        fs::remove_file(config_path)?;
    }

    if credentials_db_path.exists() {
        fs::remove_file(credentials_db_path)?;
    }

    println!("Configuration files and credentials deleted.");
    Ok(())
}

/// Setup configuration by prompting the user
fn setup_config() -> Result<Config> {
    let mut dnac_url = String::new();
    let mut username = String::new();
    let mut verify_ssl_input = String::new();

    print!("Enter Cisco DNAC URL without a / at the end (e.g., https://dnac
    io::stdout().flush()?;
    io::stdin().read_line(&mut dnac_url)?;
    dnac_url = dnac_url.trim().to_string();

    print!("Enter your username: ");
    io::stdout().flush()?;
    io::stdin().read_line(&mut username)?;
    username = username.trim().to_string();

    print!("Verify SSL certificates? (y/n): ");
    io::stdout().flush()?;
    io::stdin().read_line(&mut verify_ssl_input)?;
    let verify_ssl = verify_ssl_input.trim().to_lowercase() == "y";

    println!("Configuration complete. Please proceed to store your credenti

    Ok(Config::new(dnac_url, username, verify_ssl))
}

/// Save the configuration to a file
fn save_config(config: &Config) -> Result<()> {
    let config_path = get_config_path();
    let contents = serde_yaml::to_string(config)?;
```

```
    fs::write(config_path, contents)?;
    Ok(())
}
```

File: ./target/package/catalysh-0.0.2/src/app/update.rs

```
use std::fs;
use std::env;
use reqwest;
use serde_json::Value;
#[cfg(unix)]
use std::os::unix::fs::PermissionsExt; // Import PermissionsExt for Unix sy
#[allow(dead_code)]
pub fn update_to_latest() -> Result<(), Box<dyn std::error::Error>> {
    let repo_url = "https://api.github.com/repos/tparnell96/catalysh/releas
    let client = reqwest::blocking::Client::new();
    let response = client.get(repo_url).header("User-Agent", "Rust-App").se

    if !response.status().is_success() {
        return Err("Failed to fetch release information.".into());
    }

    let json: Value = response.json()?;
    let assets = json["assets"].as_array().ok_or("Invalid assets structure"
    let platform = if cfg!(target_os = "macos") {
        "macos"
    } else {
        "linux"
    };

    let architecture = if cfg!(target_arch = "x86_64") {
        "x86_64"
    } else if cfg!(target_arch = "aarch64") {
        "arm64"
    } else {
        "unknown"
    };

    let asset = assets.iter().find(|asset| {
        let name = asset["name"].as_str().unwrap_or("");
        name.contains(platform) && name.contains(architecture)
    }).ok_or("No compatible asset found")?;

    let download_url = asset["browser_download_url"].as_str().ok_or("Invali
    let temp_file = env::temp_dir().join(asset["name"].as_str().unwrap_or("

    println!("Downloading update...");
    let mut file = fs::File::create(&temp_file)?;
    let mut response = client.get(download_url).send()?;
```

```
    response.copy_to(&mut file)?;
    println!("Download complete: {}", temp_file.display());

    let local_bin = dirs::home_dir()
        .ok_or("Failed to determine home directory")?
        .join(".local/bin");

    if !local_bin.exists() {
        fs::create_dir_all(&local_bin)?;
    }

    let destination = local_bin.join("catalysh");

    println!("Moving binary to ~/.local/bin...");
    fs::copy(&temp_file, &destination)?;

    #[cfg(unix)]
    {
        println!("Applying executable permissions...");
        fs::set_permissions(&destination, fs::Permissions::from_mode(0o755)
    }

    println!("Update successfully applied to ~/.local/bin/catalysh.");
    Ok(())
}
```

File: ./target/package/catalysh-0.0.2/src/app/mod.rs

```
pub mod config;
pub mod update;
```

File: ./target/package/catalysh-0.0.2/src/bin/windows_installer.rs

```
use std::fs;
use std::env;
use std::path::PathBuf;
use std::process::Command;
use reqwest;
use serde_json::Value;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let repo_url = "https://api.github.com/repos/tparnell96/catsh/releases/
    let client = reqwest::blocking::Client::new();
    let response = client.get(repo_url).header("User-Agent", "Rust-App").se

    if !response.status().is_success() {
```

```
    return Err("Failed to fetch release information.".into());
}

let json: Value = response.json()?;
let assets = json["assets"].as_array().ok_or("Invalid assets structure"

// Download the binary file
let binary_asset = assets.iter().find(|asset| {
    let name = asset["name"].as_str().unwrap_or("");
    name.ends_with(".exe")
}).ok_or("No compatible binary asset found")?;
let binary_url = binary_asset["browser_download_url"]
    .as_str()
    .ok_or("Invalid binary download URL")?;
let binary_temp_file = env::temp_dir().join(binary_asset["name"].as_str

println!("Downloading binary...");
let mut file = fs::File::create(&binary_temp_file)?;
let mut response = client.get(binary_url).send()?;
response.copy_to(&mut file)?;
println!("Binary download complete: {}", binary_temp_file.display());

// Download the icon file
let icon_asset = assets.iter().find(|asset| {
    let name = asset["name"].as_str().unwrap_or("");
    name.ends_with(".ico")
}).ok_or("No compatible icon asset found")?;
let icon_url = icon_asset["browser_download_url"]
    .as_str()
    .ok_or("Invalid icon download URL")?;
let icon_temp_file = env::temp_dir().join(icon_asset["name"].as_str().u

println!("Downloading icon...");
let mut file = fs::File::create(&icon_temp_file)?;
let mut response = client.get(icon_url).send()?;
response.copy_to(&mut file)?;
println!("Icon download complete: {}", icon_temp_file.display());

// Move the binary to the application directory
let app_dir = PathBuf::from(env::var("LOCALAPPDATA")?).join("catsh");
if !app_dir.exists() {
    fs::create_dir_all(&app_dir)?;
}
let binary_destination = app_dir.join("catsh.exe");
println!("Moving binary to application directory...");
fs::copy(&binary_temp_file, &binary_destination)?;

// Move the icon to the application directory
let icon_destination = app_dir.join("catsh.ico");
println!("Moving icon to application directory...");
fs::copy(&icon_temp_file, &icon_destination)?;
```

```
    // Create the desktop shortcut
    println!("Creating desktop shortcut...");
    let desktop = dirs::desktop_dir().ok_or("Failed to locate the desktop d
    let shortcut_path = desktop.join("catsh.lnk");
    create_shortcut(&shortcut_path, &binary_destination, &icon_destination)

    println!("Update successfully applied.");
    Ok(())
}

fn create_shortcut(shortcut_path: &PathBuf, target_path: &PathBuf, icon_pat
    let output = Command::new("powershell")
        .arg("-Command")
        .arg(format!(
            r#"
            $WScript = New-Object -ComObject WScript.Shell;
            $Shortcut = $WScript.CreateShortcut('{}');
            $Shortcut.TargetPath = '{}';
            $Shortcut.IconLocation = '{}';
            $Shortcut.Save();
            "#,
            shortcut_path.display(),
            target_path.display(),
            icon_path.display()
        ))
        .output()?; // Capture output

    if !output.status.success() {
        eprintln!(
            "PowerShell failed with status: {}\nError: {}",
            output.status,
            String::from_utf8_lossy(&output.stderr)
        );
        return Err("Failed to create shortcut with icon".into());
    }

    println!(
        "PowerShell executed successfully:\n{}",
        String::from_utf8_lossy(&output.stdout)
    );

    Ok(())
}
```

File: ./target/package/catalysh-0.0.2/src/main.rs

```
mod app;
mod helpers;
mod api;
```

```
mod commands;
mod handlers;

use commands::{Cli, route_command};
use clap_repl::reedline::{DefaultPrompt, DefaultPromptSegment, FileBackedHi
use clap_repl::ClapEditor;
use dirs::home_dir;
use std::fs;
use std::path::PathBuf;

fn get_installation_dir() -> PathBuf {
    let home = home_dir().expect("Failed to determine the user's home direc
    home.join(".catalysh")
}

fn perform_first_time_installation() -> Result<(), Box<dyn std::error::Erro
    let install_dir = get_installation_dir();

    if !install_dir.exists() {
        println!("Running first-time installation...");
        fs::create_dir_all(&install_dir)?;
        fs::write(install_dir.join("version"), "1.0.0")?;
        println!("First-time installation complete.");
    }
    Ok(())
}

#[allow(non_snake_case)]
fn main() {
    env_logger::init();
    // Initial check to confirm program is correctly installed
    if let Err(e) = perform_first_time_installation() {
        eprintln!("Error during installation: {}", e);
        return;
    }

    let prompt = DefaultPrompt {
        left_prompt: DefaultPromptSegment::Basic("catalysh".to_owned()),
        ..DefaultPrompt::default()
    };

    // Create the REPL
    let rl = ClapEditor::<Cli>::builder()
        .with_prompt(Box::new(prompt))
        .with_editor_hook(|reed| {
            reed.with_history(Box::new(
                FileBackedHistory::with_file(10000, "/tmp/catalysh-cli-hist
            ))
        })
        .build();

    rl.repl(|cli| {
```

```
        route_command(cli.command);
    });
}
```

File: ./target/package/catalysh-0.0.2/src/api/clients/getclientenrichme

```
// src/api/clients/getclientenrichment.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[serde(untagged)]
pub enum StringOrNumber {
    String(String),
    Number(u64),
}

#[derive(Debug, Deserialize)]
#[serde(untagged)]
pub enum VlanId {
    String(String),
    Number(u64),
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ClientEnrichmentResponse(pub Vec<ClientEnrichment>);

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ClientEnrichment {
    pub userDetails: Option<UserDetails>,
    pub connectedDevice: Option<Vec<EnrichmentConnectedDevice>>,
    pub issueDetails: Option<IssueDetails>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct UserDetails {
    pub id: Option<String>,
    pub connectionStatus: Option<String>,
    pub tracked: Option<String>,
    pub hostType: Option<String>,
    pub userId: Option<String>,
    pub duid: Option<String>,
```

```
    pub identifier: Option<String>,
    pub hostName: Option<String>,
    pub hostOs: Option<String>,
    pub hostVersion: Option<String>,
    pub subType: Option<String>,
    pub firmwareVersion: Option<String>,
    pub deviceVendor: Option<String>,
    pub deviceForm: Option<String>,
    pub salesCode: Option<String>,
    pub countryCode: Option<String>,
    pub lastUpdated: Option<i64>,
    pub healthScore: Option<Vec<HealthScore>>,
    pub hostMac: Option<String>,
    pub hostIpV4: Option<String>,
    pub hostIpV6: Option<Vec<String>>,
    pub authType: Option<String>,
    pub vlanId: Option<VlanId>,
    pub port: Option<String>,
    pub ssid: Option<String>,
    pub frequency: Option<String>,
    pub channel: Option<String>,
    pub apGroup: Option<String>,
    pub sgt: Option<String>,
    pub location: Option<String>,
    pub clientConnection: Option<String>,
    pub connectedDevice: Option<Vec<ConnectedDevice>>,
    pub issueCount: Option<StringOrNumber>,
    pub rssi: Option<StringOrNumber>,
    pub rssiThreshold: Option<String>,
    pub rssiIsInclude: Option<String>,
    pub avgRssi: Option<String>,
    pub snr: Option<StringOrNumber>,
    pub snrThreshold: Option<String>,
    pub snrIsInclude: Option<String>,
    pub avgSnr: Option<String>,
    pub dataRate: Option<StringOrNumber>,
    pub txBytes: Option<String>,
    pub rxBytes: Option<String>,
    pub dnsResponse: Option<String>,
    pub dnsRequest: Option<String>,
    pub onboarding: Option<Onboarding>,
    // ... other fields as needed
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct HealthScore {
    pub healthType: Option<String>,
    pub reason: Option<String>,
    pub score: Option<i32>,
}
```

```rust
#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ConnectedDevice {
    #[serde(rename = "type")]
    pub type_field: Option<String>, // `type` is a reserved keyword in Rust
    pub name: Option<String>,
    pub mac: Option<String>,
    pub id: Option<String>,
    #[serde(rename = "ip address")]
    pub ip_address: Option<String>,
    pub mgmtIp: Option<String>,
    pub band: Option<String>,
    pub mode: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct EnrichmentConnectedDevice {
    pub deviceDetails: Option<DeviceDetails>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct DeviceDetails {
    pub family: Option<String>,
    #[serde(rename = "type")]
    pub type_field: Option<String>,
    pub location: Option<String>,
    pub errorCode: Option<String>,
    pub macAddress: Option<String>,
    pub role: Option<String>,
    pub apManagerInterfaceIp: Option<String>,
    pub associatedWlcIp: Option<String>,
    pub bootDateTime: Option<String>,
    pub collectionStatus: Option<String>,
    pub interfaceCount: Option<String>,
    pub lineCardCount: Option<String>,
    pub lineCardId: Option<String>,
    pub managementIpAddress: Option<String>,
    pub memorySize: Option<String>,
    pub platformId: Option<String>,
    pub reachabilityFailureReason: Option<String>,
    pub reachabilityStatus: Option<String>,
    pub snmpContact: Option<String>,
    pub snmpLocation: Option<String>,
    pub tunnelUdpPort: Option<String>,
    pub waasDeviceMode: Option<String>,
    pub series: Option<String>,
    pub inventoryStatusDetail: Option<String>,
    pub collectionInterval: Option<String>,
    pub serialNumber: Option<String>,
    pub softwareVersion: Option<String>,
```

```
    pub roleSource: Option<String>,
    pub hostname: Option<String>,
    pub upTime: Option<String>,
    pub lastUpdateTime: Option<i64>,
    pub errorDescription: Option<String>,
    pub locationName: Option<String>,
    pub tagCount: Option<String>,
    pub lastUpdated: Option<String>,
    pub instanceUuid: Option<String>,
    pub id: Option<String>,
    pub neighborTopology: Option<Vec<NeighborTopology>>,
    // ... other fields as needed
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct NeighborTopology {
    pub nodes: Option<Vec<TopologyNode>>,
    pub links: Option<Vec<TopologyLink>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct TopologyNode {
    pub role: Option<String>,
    pub name: Option<String>,
    pub id: Option<String>,
    pub description: Option<String>,
    pub deviceType: Option<String>,
    pub platformId: Option<String>,
    pub family: Option<String>,
    pub ip: Option<String>,
    pub softwareVersion: Option<String>,
    pub userId: Option<String>,
    pub nodeType: Option<String>,
    pub radioFrequency: Option<String>,
    pub clients: Option<f64>,
    pub count: Option<f64>,
    pub healthScore: Option<f64>,
    pub level: Option<f64>,
    pub fabricGroup: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct TopologyLink {
    pub source: Option<String>,
    pub linkStatus: Option<String>,
    pub label: Option<Vec<String>>,
    pub target: Option<String>,
    pub id: Option<String>,
    pub portUtilization: Option<String>,
```

```
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct Onboarding {
    pub averageRunDuration: Option<String>,
    pub maxRunDuration: Option<String>,
    pub averageAssocDuration: Option<String>,
    pub maxAssocDuration: Option<String>,
    pub averageAuthDuration: Option<String>,
    pub maxAuthDuration: Option<String>,
    pub averageDhcpDuration: Option<String>,
    pub maxDhcpDuration: Option<String>,
    pub aaaServerIp: Option<String>,
    pub dhcpServerIp: Option<String>,
    pub authDoneTime: Option<i64>,
    pub assocDoneTime: Option<i64>,
    pub dhcpDoneTime: Option<i64>,
    pub latestRootCauseList: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct IssueDetails {
    pub issue: Option<Vec<Issue>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct Issue {
    pub issueId: Option<String>,
    pub issueSource: Option<String>,
    pub issueCategory: Option<String>,
    pub issueName: Option<String>,
    pub issueDescription: Option<String>,
    pub issueEntity: Option<String>,
    pub issueEntityValue: Option<String>,
    pub issueSeverity: Option<String>,
    pub issuePriority: Option<String>,
    pub issueSummary: Option<String>,
    pub issueTimestamp: Option<i64>,
    pub suggestedActions: Option<Vec<SuggestedAction>>,
    pub impactedHosts: Option<Vec<ImpactedHost>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct SuggestedAction {
    pub message: Option<String>,
    pub steps: Option<Vec<String>>,
}
```

```rust
#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ImpactedHost {
    pub hostType: Option<String>,
    pub hostName: Option<String>,
    pub hostOs: Option<String>,
    pub ssid: Option<String>,
    pub connectedInterface: Option<String>,
    pub macAddress: Option<String>,
    pub failedAttempts: Option<i32>,
    pub location: Option<ImpactedHostLocation>,
    pub timestamp: Option<i64>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ImpactedHostLocation {
    pub siteId: Option<String>,
    pub siteType: Option<String>,
    pub area: Option<String>,
    pub building: Option<String>,
    pub floor: Option<String>,
    pub apsImpacted: Option<Vec<String>>,
}

pub async fn get_client_enrichment(
    config: &Config,
    token: &Token,
    entity_type: &str,
    entity_value: &str,
    issue_category: Option<&str>,
) -> Result<ClientEnrichmentResponse> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/client-enrichment-details", con

    // Build the request with headers
    let mut req_builder = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .header("entity_type", entity_type)
        .header("entity_value", entity_value);

    if let Some(category) = issue_category {
        req_builder = req_builder.header("issueCategory", category);
    }

    let resp = req_builder.send().await?;

    if !resp.status().is_success() {
```

```
        let status = resp.status();
        let error_text = resp.text().await.unwrap_or_default();
        return Err(anyhow!(
            "Failed to retrieve client enrichment details: {} - {}",
            status,
            error_text
        ));
    }

    let enrichment_response = resp.json::<ClientEnrichmentResponse>().await
    Ok(enrichment_response)
}
```

File: ./target/package/catalysh-0.0.2/src/api/clients/getclientdetail.rs

```
// src/api/clients/getclientdetail.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ClientDetailResponse {
    pub detail: Option<ClientDetail>,
    pub connectionInfo: Option<ConnectionInfo>,
    pub topology: Option<Topology>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ClientDetail {
    pub id: Option<String>,
    pub connectionStatus: Option<String>,
    pub hostType: Option<String>,
    pub userId: Option<String>,
    pub hostName: Option<String>,
    pub hostOs: Option<String>,
    pub hostVersion: Option<String>,
    pub subType: Option<String>,
    pub lastUpdated: Option<u64>, // Changed from Option<String> to Option<
    pub healthScore: Option<Vec<HealthScore>>,
    pub hostMac: Option<String>,
    pub hostIpV4: Option<String>,
    pub hostIpV6: Option<Vec<String>>,
    pub authType: Option<String>,
```

```
pub vlanId: Option<i32>,
pub vnid: Option<i32>,
pub ssid: Option<String>,
pub frequency: Option<String>,
pub channel: Option<String>,
pub apGroup: Option<String>,
pub location: Option<String>,
pub clientConnection: Option<String>,
pub connectedDevice: Option<Vec<ConnectedDevice>>,
pub issueCount: Option<i32>,
pub rssi: Option<String>,
pub avgRssi: Option<String>,
pub snr: Option<String>,
pub avgSnr: Option<String>,
pub dataRate: Option<String>,
pub txBytes: Option<String>,
pub rxBytes: Option<String>,
pub onboarding: Option<Onboarding>,
pub clientType: Option<String>,
pub onboardingTime: Option<u64>, // Changed from Option<String> to Opti
pub port: Option<String>,
pub iosCapable: Option<bool>,
pub tracked: Option<String>,
pub duid: Option<String>,
pub identifier: Option<String>,
pub firmwareVersion: Option<String>,
pub deviceVendor: Option<String>,
pub deviceForm: Option<String>,
pub salesCode: Option<String>,
pub countryCode: Option<String>,
pub l3VirtualNetwork: Option<String>,
pub l2VirtualNetwork: Option<String>,
pub upnId: Option<String>,
pub upnName: Option<String>,
pub sgt: Option<String>,
pub rssiThreshold: Option<String>,
pub rssiIsInclude: Option<String>,
pub snrThreshold: Option<String>,
pub snrIsInclude: Option<String>,
pub dnsResponse: Option<String>,
pub dnsRequest: Option<String>,
pub usage: Option<f64>,
pub linkSpeed: Option<f64>,
pub linkThreshold: Option<String>,
pub remoteEndDuplexMode: Option<String>,
pub txLinkError: Option<f64>,
pub rxLinkError: Option<f64>,
pub txRate: Option<f64>,
pub rxRate: Option<f64>,
pub rxRetryPct: Option<String>,
pub versionTime: Option<i64>,
pub dot11Protocol: Option<String>,
```

```
    pub slotId: Option<i32>,
    pub dot11ProtocolCapability: Option<String>,
    pub privateMac: Option<bool>,
    pub dhcpServerIp: Option<String>,
    pub aaaServerIp: Option<String>,
    pub aaaServerTransaction: Option<i32>,
    pub aaaServerFailedTransaction: Option<i32>,
    pub aaaServerSuccessTransaction: Option<i32>,
    pub aaaServerLatency: Option<f64>,
    pub aaaServerMABLatency: Option<f64>,
    pub aaaServerEAPLatency: Option<f64>,
    pub dhcpServerTransaction: Option<i32>,
    pub dhcpServerFailedTransaction: Option<i32>,
    pub dhcpServerSuccessTransaction: Option<i32>,
    pub dhcpServerLatency: Option<f64>,
    pub dhcpServerDOLatency: Option<f64>,
    pub dhcpServerRALatency: Option<f64>,
    pub maxRoamingDuration: Option<String>,
    pub upnOwner: Option<String>,
    pub connectedUpn: Option<String>,
    pub connectedUpnOwner: Option<String>,
    pub connectedUpnId: Option<String>,
    pub isGuestUPNEndpoint: Option<bool>,
    pub wlcName: Option<String>,
    pub wlcUuid: Option<String>,
    pub sessionDuration: Option<String>,
    pub intelCapable: Option<bool>,
    pub hwModel: Option<String>,
    pub powerType: Option<String>,
    pub modelName: Option<String>,
    pub bridgeVMMode: Option<String>,
    pub dhcpNakIp: Option<String>,
    pub dhcpDeclineIp: Option<String>,
    pub portDescription: Option<String>,
    pub latencyVoice: Option<f64>,
    pub latencyVideo: Option<f64>,
    pub latencyBg: Option<f64>,
    pub latencyBe: Option<f64>,
    pub trustScore: Option<String>,
    pub trustDetails: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct HealthScore {
    pub healthType: Option<String>,
    pub reason: Option<String>,
    pub score: Option<i32>,
}

#[derive(Debug, Deserialize)]
```

```rust
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ConnectedDevice {
    #[serde(rename = "type")]
    pub device_type: Option<String>,
    pub name: Option<String>,
    pub mac: Option<String>,
    pub id: Option<String>,
    #[serde(rename = "ip address")]
    pub ip_address: Option<String>,
    pub mgmtIp: Option<String>,
    pub band: Option<String>,
    pub mode: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct Onboarding {
    pub averageRunDuration: Option<String>,
    pub maxRunDuration: Option<String>,
    pub averageAssocDuration: Option<String>,
    pub maxAssocDuration: Option<String>,
    pub averageAuthDuration: Option<String>,
    pub maxAuthDuration: Option<String>,
    pub averageDhcpDuration: Option<String>,
    pub maxDhcpDuration: Option<String>,
    pub aaaServerIp: Option<String>,
    pub dhcpServerIp: Option<String>,
    pub authDoneTime: Option<u64>,   // Changed from Option<String> to Opti
    pub assocDoneTime: Option<u64>,  // Changed from Option<String> to Opti
    pub dhcpDoneTime: Option<u64>,   // Changed from Option<String> to Opti
    pub assocRootcauseList: Option<Vec<String>>,
    pub aaaRootcauseList: Option<Vec<String>>,
    pub dhcpRootcauseList: Option<Vec<String>>,
    pub otherRootcauseList: Option<Vec<String>>,
    pub latestRootCauseList: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ConnectionInfo {
    pub hostType: Option<String>,
    pub nwDeviceName: Option<String>,
    pub nwDeviceMac: Option<String>,
    pub protocol: Option<String>,
    pub band: Option<String>,
    pub spatialStream: Option<String>,
    pub channel: Option<String>,
    pub channelWidth: Option<String>,
    pub wmm: Option<String>,
```

```
    pub uapsd: Option<String>,
    pub timestamp: Option<u64>, // Changed from Option<String> to Option<u6
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct Topology {
    pub nodes: Option<Vec<TopologyNode>>,
    pub links: Option<Vec<TopologyLink>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyNode {
    pub role: Option<String>,
    pub name: Option<String>,
    pub id: Option<String>,
    pub description: Option<String>,
    pub deviceType: Option<String>,
    pub platformId: Option<String>,
    pub family: Option<String>,
    pub ip: Option<String>,
    pub softwareVersion: Option<String>,
    pub userId: Option<String>,
    pub nodeType: Option<String>,
    pub radioFrequency: Option<String>,
    pub clients: Option<f64>,
    pub count: Option<i32>,
    pub healthScore: Option<f64>,
    pub level: Option<f64>,
    pub fabricGroup: Option<String>,
    pub connectedDevice: Option<String>,
    pub fabricRole: Option<Vec<String>>,
    pub stackType: Option<String>,
    pub ipv6: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyLink {
    pub source: Option<String>,
    pub linkStatus: Option<String>,
    pub label: Option<Vec<String>>,
    pub target: Option<String>,
    pub id: Option<String>,
    pub portUtilization: Option<f64>,
}

pub async fn get_client_detail(
```

```
    config: &Config,
    token: &Token,
    mac_address: &str,
) -> Result<ClientDetailResponse> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/client-detail", config.dnac_url

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .query(&[("macAddress", mac_address)])
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve client details: {}",
            resp.status()
        ));
    }

    let client_detail_response = resp.json::<ClientDetailResponse>().await?
    Ok(client_detail_response)
}
```

File: ./target/package/catalysh-0.0.2/src/api/clients/mod.rs

```
pub mod getclientdetail;
pub mod getclientenrichment;
```

File: ./target/package/catalysh-0.0.2/src/api/mod.rs

```
pub mod authentication;
pub mod devices;
pub mod clients;
pub mod issues;
pub mod wireless;
```

File: ./target/package/catalysh-0.0.2/src/api/wireless/mod.rs

```
// src/api/wireless/mod.rs

pub mod accesspointconfig;
```

File: ./target/package/catalysh-0.0.2/src/api/wireless/accesspointcon

```
// src/api/wireless/getaccesspointconfig.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ApConfig {
    pub instanceUuid: Option<serde_json::Value>,
    pub instanceId: Option<f64>,
    pub authEntityId: Option<serde_json::Value>,
    pub displayName: Option<String>,
    pub authEntityClass: Option<serde_json::Value>,
    pub instanceTenantId: Option<String>,
    pub _orderedListOEIndex: Option<f64>,
    pub _orderedListOEAssocName: Option<serde_json::Value>,
    pub _creationOrderIndex: Option<f64>,
    pub _isBeingChanged: Option<bool>,
    pub deployPending: Option<String>,
    pub instanceCreatedOn: Option<serde_json::Value>,
    pub instanceUpdatedOn: Option<serde_json::Value>,
    pub changeLogList: Option<serde_json::Value>,
    pub instanceOrigin: Option<serde_json::Value>,
    pub lazyLoadedEntities: Option<serde_json::Value>,
    pub instanceVersion: Option<f64>,
    pub adminStatus: Option<String>,
    pub apHeight: Option<f64>,
    pub apMode: Option<String>,
    pub apName: Option<String>,
    pub ethMac: Option<String>,
    pub failoverPriority: Option<String>,
    pub ledBrightnessLevel: Option<i64>,
    pub ledStatus: Option<String>,
    pub location: Option<String>,
    pub macAddress: Option<String>,
    pub primaryControllerName: Option<String>,
    pub primaryIpAddress: Option<String>,
    pub secondaryControllerName: Option<String>,
    pub secondaryIpAddress: Option<String>,
    pub tertiaryControllerName: Option<String>,
    pub tertiaryIpAddress: Option<String>,
    pub meshDTOs: Option<Vec<serde_json::Value>>,
    pub radioDTOs: Option<Vec<RadioDTO>>,
    pub internalKey: Option<InternalKey>,
}
```

```rust
#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct RadioDTO {
    pub instanceUuid: Option<serde_json::Value>,
    pub instanceId: Option<f64>,
    pub authEntityId: Option<serde_json::Value>,
    pub displayName: Option<String>,
    pub authEntityClass: Option<serde_json::Value>,
    pub instanceTenantId: Option<String>,
    pub _orderedListOEIndex: Option<f64>,
    pub _orderedListOEAssocName: Option<serde_json::Value>,
    pub _creationOrderIndex: Option<f64>,
    pub _isBeingChanged: Option<bool>,
    pub deployPending: Option<String>,
    pub instanceCreatedOn: Option<serde_json::Value>,
    pub instanceUpdatedOn: Option<serde_json::Value>,
    pub changeLogList: Option<serde_json::Value>,
    pub instanceOrigin: Option<serde_json::Value>,
    pub lazyLoadedEntities: Option<serde_json::Value>,
    pub instanceVersion: Option<f64>,
    pub adminStatus: Option<String>,
    pub antennaAngle: Option<f64>,
    pub antennaElevAngle: Option<f64>,
    pub antennaGain: Option<i64>,
    pub antennaPatternName: Option<String>,
    pub channelAssignmentMode: Option<String>,
    pub channelNumber: Option<i64>,
    pub channelWidth: Option<String>,
    pub cleanAirSI: Option<String>,
    pub ifType: Option<i64>,
    pub ifTypeValue: Option<String>,
    pub macAddress: Option<String>,
    pub powerAssignmentMode: Option<String>,
    pub powerlevel: Option<i64>,
    pub radioBand: Option<serde_json::Value>,
    pub radioRoleAssignment: Option<serde_json::Value>,
    pub slotId: Option<i64>,
    pub internalKey: Option<InternalKey>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct InternalKey {
    #[serde(rename = "type")]
    pub type_field: Option<String>,
    pub id: Option<f64>,
    pub longType: Option<String>,
    pub url: Option<String>,
}
```

```
pub async fn get_ap_config(
    config: &Config,
    token: &Token,
    mac_address: &str,
) -> Result<ApConfig> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/wireless/accesspoint-configurat

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .query(&[("key", mac_address)])
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve AP config: {}",
            resp.status()
        ));
    }

    let ap_config = resp.json::<ApConfig>().await?;
    Ok(ap_config)
}
```

File: ./target/package/catalysh-0.0.2/src/api/issues/getissuelist.rs

```
// src/api/issues/getissuelist.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;
use std::collections::HashMap;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct IssueListResponse {
    pub version: Option<String>,
    pub totalCount: Option<String>,
    pub response: Option<Vec<Issue>>,
}

#[derive(Debug, Deserialize)]
```

```rust
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct Issue {
    pub issueId: Option<String>,
    pub name: Option<String>,
    pub siteId: Option<String>,
    pub deviceId: Option<String>,
    pub deviceRole: Option<String>,
    pub aiDriven: Option<String>,
    pub clientMac: Option<String>,
    pub issue_occurence_count: Option<i32>,
    pub status: Option<String>,
    pub priority: Option<String>,
    pub category: Option<String>,
    pub last_occurence_time: Option<i64>,
}

pub async fn get_issue_list(
    config: &Config,
    token: &Token,
    search_params: &HashMap<String, String>,
) -> Result<IssueListResponse> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/issues", config.dnac_url);

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .query(&search_params)
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve issue list: {}",
            resp.status()
        ));
    }

    let issue_list_response = resp.json::<IssueListResponse>().await?;
    Ok(issue_list_response)
}
```

File: ./target/package/catalysh-0.0.2/src/api/issues/mod.rs

```
pub mod getissuelist;
```

File: ./target/package/catalysh-0.0.2/src/api/authentication/auth.rs

```
use crate::app::config::Config;
use crate::helpers::utils;
use anyhow::{anyhow, Result};
use argon2::{
    password_hash::{PasswordHasher, PasswordVerifier, SaltString},
    Argon2,
};
use rand::rngs::OsRng;
use rusqlite::{params, Connection};
use std::fs;
use std::path::PathBuf;

use reqwest::Client;
use serde::Deserialize;

#[derive(Deserialize)]
#[allow(non_snake_case)]
struct TokenResponse {
    Token: String,
}

#[derive(Clone)]
#[allow(non_snake_case)]
pub struct Token {
    pub value: String,
    pub obtained_at: u64,
    pub expires_at: u64,
}



pub async fn authenticate(config: &Config) -> Result<Token> {
    // Check for existing token
    if let Some(token) = load_token()? {
        if token.expires_at > utils::current_timestamp() {
            // Token is still valid
            return Ok(token);
        }
    }

    // Token is missing or expired; proceed to authenticate
    let credentials = load_credentials(&config.username)?;
    let password = prompt_password(&config.username)?;
```

```
    if !verify_password(&password, &credentials.password_hash)? {
        return Err(anyhow!("Invalid password"));
    }

    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let auth_url = format!("{}/dna/system/api/v1/auth/token", config.dnac_u

    let resp = client
        .post(&auth_url)
        .basic_auth(&config.username, Some(&password))
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Authentication failed with status: {}",
            resp.status()
        ));
    }

    let token_resp: TokenResponse = resp.json().await?;

    let obtained_at = utils::current_timestamp();
    let expires_at = obtained_at + 1 * 60 * 60; // Token valid for 1 hour

    let token = Token {
        value: token_resp.Token,
        obtained_at,
        expires_at,
    };

    // Store the token
    store_token(&token)?;

    Ok(token)
}

struct StoredCredentials {
    password_hash: String,
}

fn get_db_path() -> PathBuf {
    let mut db_path = dirs::config_dir().unwrap();
    db_path.push("catsh");
    db_path.push("credentials.db");
    db_path
}

fn load_credentials(username: &str) -> Result<StoredCredentials> {
```

```
    let db_path = get_db_path();
    if !db_path.exists() {
        println!("No credentials database found. Starting setup...");
        store_credentials(username)?;
    }

    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    let mut stmt = conn.prepare("SELECT password_hash FROM credentials WHER
    let mut rows = stmt.query(params![username])?;

    if let Some(row) = rows.next()? {
        let password_hash: String = row.get(0)?;
        Ok(StoredCredentials { password_hash })
    } else {
        println!("No credentials found for user '{}'. Starting setup...", u
        store_credentials(username)?;

        // After storing credentials, try to load them again
        let mut stmt = conn.prepare("SELECT password_hash FROM credentials
        let mut rows = stmt.query(params![username])?;

        if let Some(row) = rows.next()? {
            let password_hash: String = row.get(0)?;
            Ok(StoredCredentials { password_hash })
        } else {
            Err(anyhow!("Credentials not found after setup"))
        }
    }
}

fn store_credentials(username: &str) -> Result<()> {
    let password = prompt_new_password(username)?;

    let db_path = get_db_path();
    fs::create_dir_all(db_path.parent().unwrap())?;
    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    let salt = SaltString::generate(&mut OsRng);
    let argon2 = Argon2::default();
    let password_hash = argon2
        .hash_password(password.as_bytes(), &salt)
        .map_err(|e| anyhow!(e))?
        .to_string();

    conn.execute(
        "INSERT INTO credentials (username, password_hash) VALUES (?1, ?2)"
        params![username, password_hash],
```

```
    )?;

    Ok(())
}

fn prompt_password(username: &str) -> Result<String> {
    let password = rpassword::prompt_password(format!("Enter password for {
    Ok(password)
}

fn prompt_new_password(username: &str) -> Result<String> {
    let password = rpassword::prompt_password(format!("Set a new password f
    let confirm_password = rpassword::prompt_password("Confirm password: ")

    if password != confirm_password {
        return Err(anyhow!("Passwords do not match"));
    }

    Ok(password)
}

fn verify_password(password: &str, password_hash: &str) -> Result<bool> {
    let parsed_hash = argon2::password_hash::PasswordHash::new(password_has
        .map_err(|e| anyhow!(e))?;
    let argon2 = Argon2::default();
    Ok(argon2
        .verify_password(password.as_bytes(), &parsed_hash)
        .is_ok())
}

// Functions to store and load the token
fn store_token(token: &Token) -> Result<()> {
    let db_path = get_db_path();
    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    conn.execute(
        "DELETE FROM token", // Clear any existing token
        [],
    )?;

    conn.execute(
        "INSERT INTO token (value, obtained_at, expires_at) VALUES (?1, ?2,
        params![token.value, token.obtained_at, token.expires_at],
    )?;

    Ok(())
}

fn load_token() -> Result<Option<Token>> {
    let db_path = get_db_path();
```

```
    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    let mut stmt = conn.prepare("SELECT value, obtained_at, expires_at FROM
    let mut rows = stmt.query([])?;

    if let Some(row) = rows.next()? {
        let value: String = row.get(0)?;
        let obtained_at: u64 = row.get(1)?;
        let expires_at: u64 = row.get(2)?;

        Ok(Some(Token {
            value,
            obtained_at,
            expires_at,
        }))
    } else {
        Ok(None)
    }
}

fn create_tables(conn: &Connection) -> Result<()> {
    conn.execute(
        "CREATE TABLE IF NOT EXISTS credentials (
            id INTEGER PRIMARY KEY,
            username TEXT NOT NULL,
            password_hash TEXT NOT NULL
        )",
        [],
    )?;

    conn.execute(
        "CREATE TABLE IF NOT EXISTS token (
            id INTEGER PRIMARY KEY,
            value TEXT NOT NULL,
            obtained_at INTEGER NOT NULL,
            expires_at INTEGER NOT NULL
        )",
        [],
    )?;

    Ok(())
}
```

File: ./target/package/catalysh-0.0.2/src/api/authentication/mod.rs

```
pub mod auth;
```

File: ./target/package/catalysh-0.0.2/src/api/devices/getdevicelist.rs

```
use crate::api::authentication::auth::{self, Token};
use crate::app::config::Config;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[serde(rename_all = "camelCase")]
#[allow(dead_code)]
pub struct AllDevices {
    pub reachability_failure_reason: Option<String>,
    pub reachability_status: Option<String>,
    pub series: Option<String>,
    pub snmp_contact: Option<String>,
    pub snmp_location: Option<String>,
    pub tag_count: Option<String>,
    pub tunnel_udp_port: Option<serde_json::Value>, // Use `serde_json::Val
    pub uptime_seconds: Option<i64>, // Assuming "integer" corresponds to i
    pub waas_device_mode: Option<serde_json::Value>,
    pub serial_number: Option<String>,
    pub last_update_time: Option<i64>,
    pub mac_address: Option<String>,
    pub up_time: Option<String>,
    pub device_support_level: Option<String>,
    pub hostname: Option<String>,
    pub device_type: Option<String>, // Renamed "type" to "device_type" to
    pub memory_size: Option<String>,
    pub family: Option<String>,
    pub error_code: Option<String>,
    pub software_type: Option<String>,
    pub software_version: Option<String>,
    pub description: Option<String>,
    pub role_source: Option<String>,
    pub location: Option<serde_json::Value>,
    pub role: Option<String>,
    pub collection_interval: Option<String>,
    pub inventory_status_detail: Option<String>,
    pub ap_ethernet_mac_address: Option<String>,
    pub ap_manager_interface_ip: Option<String>,
    pub associated_wlc_ip: Option<String>,
    pub boot_date_time: Option<String>,
    pub collection_status: Option<String>,
    pub error_description: Option<String>,
    pub interface_count: Option<String>,
    pub last_updated: Option<String>,
    pub line_card_count: Option<String>,
    pub line_card_id: Option<String>,
    pub location_name: Option<serde_json::Value>,
    pub managed_atleast_once: Option<bool>,
```

```
    pub management_ip_address: Option<String>,
    pub platform_id: Option<String>,
    pub management_state: Option<String>,
    pub instance_tenant_id: Option<String>,
    pub instance_uuid: Option<String>,
    pub id: Option<String>,
}

#[derive(Debug, Deserialize)]
struct DevicesResponse {
    response: Vec<AllDevices>,
}

pub async fn get_all_devices(config: &Config, token: &Token) -> Result<Vec<
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let mut all_devices: Vec<AllDevices> = Vec::new();
    let mut offset = 1; // Adjust based on API documentation (could be 0)
    let limit = 500;    // Set the limit as per API maximum

    loop {
        let devices_url = format!(
            "{}/dna/intent/api/v1/network-device?offset={}&limit={}",
            config.dnac_url, offset, limit
        );

        // Perform the API request with reauthentication handling
        let devices_response: DevicesResponse =
            send_authenticated_request(&client, config, token, &devices_url

        let devices = devices_response.response;

        if devices.is_empty() {
            // No more devices to fetch
            break;
        }

        all_devices.extend(devices);

        // Increment offset
        offset += limit;
    }

    Ok(all_devices)
}

/// Sends an authenticated GET request, handling reauthentication if necess
async fn send_authenticated_request<T: serde::de::DeserializeOwned>(
    client: &Client,
    config: &Config,
```

```
    token: &Token,
    url: &str,
) -> Result<T> {
    let mut current_token = token.clone();

    loop {
        let resp = client
            .get(url)
            .header("X-Auth-Token", &current_token.value)
            .send()
            .await?;

        if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
            // Token expired or invalid, reauthenticate and retry
            eprintln!("Token expired. Reauthenticating...");
            current_token = auth::authenticate(config).await?;
            continue; // Retry the request with the new token
        }

        if !resp.status().is_success() {
            return Err(anyhow!(
                "Failed to complete request: {}",
                resp.status()
            ));
        }

        // Deserialize and return the response
        let result = resp.json::<T>().await?;
        return Ok(result);
    }
}
```

File: ./target/package/catalysh-0.0.2/src/api/devices/devicedetailenri

```
// src/api/devices/devicedetailenrichment.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct DeviceEnrichmentResponse {
    pub deviceDetails: DeviceDetails,
}

#[derive(Debug, Deserialize)]
```

```rust
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct DeviceDetails {
    pub family: Option<String>,
    #[serde(rename = "type")]
    pub type_field: Option<String>,
    pub location: Option<serde_json::Value>,
    pub errorCode: Option<String>,
    pub macAddress: Option<String>,
    pub role: Option<String>,
    pub apManagerInterfaceIp: Option<String>,
    pub associatedWlcIp: Option<String>,
    pub bootDateTime: Option<String>,
    pub collectionStatus: Option<String>,
    pub interfaceCount: Option<String>,
    pub lineCardCount: Option<String>,
    pub lineCardId: Option<String>,
    pub managementIpAddress: Option<String>,
    pub memorySize: Option<String>,
    pub platformId: Option<String>,
    pub reachabilityFailureReason: Option<String>,
    pub reachabilityStatus: Option<String>,
    pub snmpContact: Option<String>,
    pub snmpLocation: Option<String>,
    pub tunnelUdpPort: Option<serde_json::Value>,
    pub waasDeviceMode: Option<serde_json::Value>,
    pub series: Option<String>,
    pub inventoryStatusDetail: Option<String>,
    pub collectionInterval: Option<String>,
    pub serialNumber: Option<String>,
    pub softwareVersion: Option<String>,
    pub roleSource: Option<String>,
    pub hostname: Option<String>,
    pub upTime: Option<String>,
    pub lastUpdateTime: Option<i64>,
    pub errorDescription: Option<String>,
    pub locationName: Option<serde_json::Value>,
    pub tagCount: Option<String>,
    pub lastUpdated: Option<String>,
    pub instanceUuid: Option<String>,
    pub id: Option<String>,
    pub neighborTopology: Option<Vec<NeighborTopology>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct NeighborTopology {
    pub nodes: Option<Vec<TopologyNode>>,
    pub links: Option<Vec<TopologyLink>>,
}
```

```rust
#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyNode {
    pub role: Option<String>,
    pub name: Option<String>,
    pub id: Option<String>,
    pub description: Option<String>,
    pub deviceType: Option<String>,
    pub platformId: Option<String>,
    pub family: Option<String>,
    pub ip: Option<String>,
    pub softwareVersion: Option<String>,
    pub userId: Option<serde_json::Value>,
    pub nodeType: Option<String>,
    pub radioFrequency: Option<serde_json::Value>,
    pub clients: Option<serde_json::Value>,
    pub count: Option<serde_json::Value>,
    pub healthScore: Option<i32>,
    pub level: Option<f64>,
    pub fabricGroup: Option<serde_json::Value>,
    pub connectedDevice: Option<serde_json::Value>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyLink {
    pub source: Option<String>,
    pub linkStatus: Option<String>,
    pub label: Option<Vec<String>>,
    pub target: Option<String>,
    pub id: Option<serde_json::Value>,
    pub portUtilization: Option<serde_json::Value>,
}

pub async fn get_device_enrichment(
    config: &Config,
    token: &Token,
    entity_type: &str,
    entity_value: &str,
) -> Result<DeviceDetails> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/device-enrichment-details", con

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .header("entity_type", entity_type)
```

```
        .header("entity_value", entity_value)
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve device enrichment details: {}",
            resp.status()
        ));
    }

    let enrichment_responses = resp.json::<Vec<DeviceEnrichmentResponse>>()

    if let Some(first_response) = enrichment_responses.into_iter().next() {
        Ok(first_response.deviceDetails)
    } else {
        Err(anyhow!("No device enrichment details found"))
    }
}
```

File: ./target/package/catalysh-0.0.2/src/api/devices/mod.rs

```
// src/api/devices/mod.rs

pub mod getdevicelist;
pub mod devicedetailenrichment;
```

File: ./target/package/catalysh-0.0.2/src/templates/exampleintegratio

```
fn handle_example(command: ExampleSubcommands) {
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        let token = match api::auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
                error!("Authentication failed: {}", e);
                return;
            }
        };
```

```
        if let Err(e) = handle_example_command(config, token, command).awai
            error!("Error handling example command: {}", e);
        }
    });
}
```

File: ./target/package/catalysh-0.0.2/src/templates/templates.rs

```
use clap::{Args, Parser, Subcommand};
use reqwest::{Client, Method};
use serde::{Deserialize, Serialize};
use anyhow::{Result, anyhow};
use crate::config::Config;
use crate::api::auth::Token;

// Root command template
#[derive(Debug, Parser)]
#[command(name = "catsh", about = "A REPL CLI for managing network devices"
pub struct Cli {
    #[command(subcommand)]
    pub command: Commands,
}

// Main command structure
#[derive(Debug, Subcommand)]
pub enum Commands {
    ExampleCommand {
        #[command(subcommand)]
        subcommand: ExampleSubcommands,
    },
}

// Subcommands template
#[derive(Debug, Subcommand)]
pub enum ExampleSubcommands {
    List,
    Details {
        #[arg(help = "ID of the item to retrieve details for")]
        id: String,
    },
    Create(NewItemArgs),
}

// Arguments for a subcommand
#[derive(Debug, Args)]
pub struct NewItemArgs {
    #[arg(help = "Name of the new item")]
    pub name: String,
    #[arg(help = "Description of the new item")]
    pub description: String,
```

```rust
}

// Handle ExampleCommand
pub async fn handle_example_command(config: Config, token: Token, subcomman
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    match subcommand {
        ExampleSubcommands::List => {
            let response = api_get::<Vec<Item>>(&client, &config, &token, "
            println!("Items: {:?}", response);
        }
        ExampleSubcommands::Details { id } => {
            let response = api_get::<Item>(&client, &config, &token, &forma
            println!("Item Details: {:?}", response);
        }
        ExampleSubcommands::Create(args) => {
            let new_item = NewItem {
                name: args.name,
                description: args.description,
            };
            let response = api_post::<Item, NewItem>(&client, &config, &tok
            println!("Created Item: {:?}", response);
        }
    }

    Ok(())
}

// Example Data Structures
#[derive(Debug, Deserialize)]
pub struct Item {
    pub id: String,
    pub name: String,
    pub description: String,
}

#[derive(Debug, Serialize)]
pub struct NewItem {
    pub name: String,
    pub description: String,
}

// API Call Templates
pub async fn api_get<T: Deserialize<'static>>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
```

```
    let mut resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .send()
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("GET request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}

pub async fn api_post<T: Deserialize<'static>, U: Serialize>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
    body: &U,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .post(&url)
        .header("X-Auth-Token", &token.value)
        .json(body)
        .send()
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("POST request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}

pub async fn api_put<T: Deserialize<'static>, U: Serialize>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
    body: &U,
) -> Result<T> {
```

```
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .put(&url)
        .header("X-Auth-Token", &token.value)
        .json(body)
        .send()
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("PUT request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}

pub async fn api_delete<T: Deserialize<'static>>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .delete(&url)
        .header("X-Auth-Token", &token.value)
        .send()
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("DELETE request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}
```

File: ./target/package/catalysh-0.0.2/src/commands/app/config.rs

```
use clap::Subcommand;

#[derive(Debug, Subcommand)]
```

```
pub enum AppConfigCommands {
    /// Reset app configuration
    Reset,
}
```

File: ./target/package/catalysh-0.0.2/src/commands/app/update.rs

```
// Empty file since `Update` doesn't have subcommands
```

File: ./target/package/catalysh-0.0.2/src/commands/app/mod.rs

```
pub mod config;
pub mod update; // Added this line

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum AppCommands {
    /// App configuration commands
    Config {
        #[command(subcommand)]
        subcommand: config::AppConfigCommands,
    },
    /// Update the program to the latest release available (Program restart
    Update,
    // Additional app-related subcommands can be added here
}
```

File: ./target/package/catalysh-0.0.2/src/commands/config/comman

File: ./target/package/catalysh-0.0.2/src/commands/config/mod.rs

```
// This module is currently empty since the config command starts a sub-REP
```

File: ./target/package/catalysh-0.0.2/src/commands/show/ap.rs

```
// src/commands/show/ap.rs

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum ApCommands {
    /// Show AP configuration by MAC address
    Config {
        /// MAC address of the AP
        mac_address: String,
    },
}
```

File: ./target/package/catalysh-0.0.2/src/commands/show/device.rs

```
// src/commands/show/device.rs

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum DeviceCommands {
    /// List devices
    List {
        #[command(subcommand)]
        filter: DeviceListFilter,
    },
    /// Show device details
    Detail {
        #[command(subcommand)]
        filter: DeviceDetailFilter,
    },
    /// Show device enrichment detail
    Enrichment {
        #[command(subcommand)]
        filter: DeviceEnrichmentFilter,
    },
}

#[derive(Debug, Subcommand)]
pub enum DeviceListFilter {
    /// List all devices
    All,
    /// List devices filtered by hostname
    Hostname {
        /// Optional partial hostname to filter by
        partial_hostname: Option<String>,
    },
    /// List devices filtered by IP address
```

```
    Ip {
        /// Optional partial IP address to filter by
        partial_ip: Option<String>,
    },
    /// List devices filtered by WLC IP address
    Wlc {
        /// Optional partial WLC ip to filter by
        partial_wlc: Option<String>,
    },
}

#[derive(Debug, Subcommand)]
pub enum DeviceDetailFilter {
    /// Show device detail by hostname
    Hostname {
        /// The hostname of the device
        hostname: String,
    },
    /// Show device detail by MAC address
    Mac {
        /// The MAC address of the device
        mac_address: String,
    },
    /// Show device detail by IP address
    Ip {
        /// The IP address of the device
        ip_address: String,
    },
}

#[derive(Debug, Subcommand)]
pub enum DeviceEnrichmentFilter {
    /// Enrichment by MAC address
    Mac {
        /// The MAC address of the device
        mac_address: String,
    },
    /// Enrichment by IP address
    Ip {
        /// The IP address of the device
        ip_address: String,
    },
}
```

File: ./target/package/catalysh-0.0.2/src/commands/show/client.rs

```
// src/commands/show/client.rs

use clap::Subcommand;
```

```
#[derive(Debug, Subcommand)]
pub enum ClientCommands {
    /// Show client details by MAC address
    Detail {
        /// MAC address of the client
        mac_address: String,
    },
    /// Show client enrichment by network user ID or MAC address
    Enrichment {
        /// The entity type (network_user_id or mac_address)
        #[arg(value_parser = ["network_user_id", "mac_address"])]
        entity_type: String,
        /// The value of the entity (user ID or MAC address)
        entity_value: String,
        /// Optional issue category
        #[arg(long)]
        issue_category: Option<String>,
    },
}
```

File: ./target/package/catalysh-0.0.2/src/commands/show/mod.rs

```
pub mod device;
pub mod client;
pub mod issue;
pub mod ap;

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum ShowCommands {
    /// Show device information
    Device {
        #[command(subcommand)]
        subcommand: device::DeviceCommands,
    },
    /// Show client information
    Client {
        #[command(subcommand)]
        subcommand: client::ClientCommands,
    },
    /// Show issues in Catalyst Center
    Issue {
        #[command(subcommand)]
        subcommand: issue::IssueCommands,
    },
    /// Show Access Point information
    Ap {
        #[command(subcommand)]
        subcommand: ap::ApCommands,
```

```
    },
}
```

File: ./target/package/catalysh-0.0.2/src/commands/show/issue.rs

```
// src/commands/show/issue.rs

#[allow(unused_imports)]
use clap::{Parser, Subcommand, ValueEnum};

#[derive(Debug, Subcommand)]
#[allow(unused_imports)]
pub enum IssueCommands {
    /// List issues based on search criteria
    List {
        /// Search option (e.g., deviceId, macAddress, priority, etc.)
        #[arg(value_enum)]
        search_option: Option<SearchOption>,
        /// Search input corresponding to the search option
        search_input: Option<String>,
    },
}

#[derive(Debug, Clone, ValueEnum)]
#[allow(dead_code)]
pub enum SearchOption {
    /// Start time to search from when looking for issues
    StartTime,
    /// End time used in conjunction with StartTime
    EndTime,
    /// SiteID gotten from a show site detail command
    SiteId,
    /// DeviceID gotten from a show device detail command
    DeviceId,
    /// MAC Address of a device or client
    MacAddress,
    /// One of these options - P1, P2, P3, P4
    Priority,
    /// Only pull issues are/aren't AI Driven - must be "Yes" or "No"
    AiDriven,
    /// Only pull issues with a specific status
    IssueStatus,
}
```

File: ./target/package/catalysh-0.0.2/src/commands/mod.rs

```
pub mod show;
pub mod config;
pub mod app;

use clap::{Parser, Subcommand};
use crate::handlers::{handle_show_command, handle_config_command, handle_ap

#[derive(Debug, Parser)]
#[command(name = "catsh", about = "A command line interface for Cisco Catal
pub struct Cli {
    #[command(subcommand)]
    pub command: Commands,
}

#[derive(Debug, Subcommand)]
pub enum Commands {
    /// Show commands
    Show {
        #[command(subcommand)]
        subcommand: show::ShowCommands,
    },
    /// Start configuration sub-REPL
    Config,
    /// App-specific commands
    App {
        #[command(subcommand)]
        subcommand: app::AppCommands,
    },
    /// Exit the program
    Exit,
}

pub fn route_command(command: Commands) {
    match command {
        Commands::Show { subcommand } => handle_show_command(subcommand),
        Commands::Config => handle_config_command(),
        Commands::App { subcommand } => handle_app_command(subcommand),
        Commands::Exit => {
            println!("Exiting catsh...");
            std::process::exit(0);
        }
    }
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/app/config.rs

```
use log::error;
use crate::app::config; // Adjusted import
use crate::commands::app::config::AppConfigCommands;

pub fn handle_app_config_command(subcommand: AppConfigCommands) {
    match subcommand {
        AppConfigCommands::Reset => {
            if let Err(e) = config::reset_config() {
                error!("Failed to reset configuration: {}", e);
            } else {
                println!("Configuration reset successfully.");
            }
        }
    }
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/app/update.rs

```
#[allow(unused_imports)]
use crate::app::update; // Corrected import

pub fn handle_update_command() {
    #[cfg(any(target_os = "linux", target_os = "macos"))]
    {
        if let Err(e) = update::update_to_latest() {
            eprintln!("Update failed: {}", e);
        } else {
            println!("Update completed successfully.");
        }
    }

    #[cfg(target_os = "windows")]
    {
        println!("Please download and run the latest `windows_installer.exe
    }
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/app/mod.rs

```
pub mod config;
pub mod update;

use crate::commands::app::AppCommands;
```

```
pub fn handle_app_command(subcommand: AppCommands) {
    match subcommand {
        AppCommands::Config { subcommand } => config::handle_app_config_com
        AppCommands::Update => update::handle_update_command(),
        // Handle other app subcommands here
    }
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/config/repl.rs

```
use clap_repl::reedline::{DefaultPrompt, DefaultPromptSegment, FileBackedHi
use clap_repl::ClapEditor;
use clap::{Parser, Subcommand};

#[derive(Debug, Parser)]
#[command(name = "", about = "Configuration Mode REPL")]
struct ConfigCli {
    #[command(subcommand)]
    command: ConfigCommands,
}

#[derive(Debug, Subcommand)]
enum ConfigCommands {
    /// Dummy command for demonstration
    Dummy,
    /// Exit configuration mode
    Exit,
    /// End configuration mode
    End,
}

pub fn start_config_repl() {
    println!("Entering configuration mode. Type 'exit' or 'end' to leave.")
    let prompt = DefaultPrompt {
        left_prompt: DefaultPromptSegment::Basic("catsh(config)#".to_owned(
        ..DefaultPrompt::default()
    };

    let rl = ClapEditor::<ConfigCli>::builder()
        .with_prompt(Box::new(prompt))
        .with_editor_hook(|reed| {
            reed.with_history(Box::new(
                FileBackedHistory::with_file(10000, "/tmp/catsh-config-cli-
            ))
        })
        .build();

    rl.repl(|cli| {
        match cli.command {
```

```
            ConfigCommands::Dummy => {
                println!("Dummy command executed in config mode.");
            }
            ConfigCommands::Exit | ConfigCommands::End => {
                println!("Exiting configuration mode.");
                std::process::exit(0);
            }
        }
    });
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/config/mod.rs

```
pub mod repl;

pub fn handle_config_command() {
    repl::start_config_repl();
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/show/ap.rs

```
// src/handlers/show/ap.rs

use crate::commands::show::ap::ApCommands;
use crate::app::config;
use crate::api::authentication::auth;
use crate::api::wireless::accesspointconfig;
use crate::helpers::utils;
use log::error;

pub fn handle_ap_command(subcommand: ApCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        // Load configuration
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        // Authenticate and get token
        let token = match auth::authenticate(&config).await {
            Ok(t) => t,
```

```
            Err(e) => {
                error!("Authentication failed: {}", e);
                return;
            }
        };

        match subcommand {
            ApCommands::Config { mac_address } => {
                // Fetch AP config
                match accesspointconfig::get_ap_config(&config, &token, &ma
                    Ok(ap_config) => {
                        utils::print_ap_config(ap_config);
                    }
                    Err(e) => {
                        error!("Failed to retrieve AP config: {}", e);
                    }
                }
            }
        }
    });
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/show/device.rs

```
// src/handlers/show/device.rs

use log::error;
use crate::app::config;
use crate::helpers::utils;
use crate::api::authentication::auth;
use crate::api::devices::{devicedetailenrichment, getdevicelist};
use crate::commands::show::device::{
    DeviceCommands, DeviceDetailFilter, DeviceEnrichmentFilter, DeviceListF
};

pub fn handle_device_command(subcommand: DeviceCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        let token = match auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
```

```
            error!("Authentication failed: {}", e);
            return;
        }
    };

    match subcommand {
        DeviceCommands::List { filter } => {
            // Fetch all devices
            match getdevicelist::get_all_devices(&config, &token).await
                Ok(devices) => {
                    // Apply filter if necessary
                    let filtered_devices = match filter {
                        DeviceListFilter::All => devices,
                        DeviceListFilter::Hostname { partial_hostname } 
                            devices
                                .into_iter()
                                .filter(|device| {
                                    if let Some(ref name) = device.host
                                        if let Some(ref partial) = part
                                            name.contains(partial)
                                        } else {
                                            true // Include all devices
                                        }
                                    } else {
                                        false
                                    }
                                })
                                .collect()
                        }
                        DeviceListFilter::Ip { partial_ip } => {
                            devices
                                .into_iter()
                                .filter(|device| {
                                    if let Some(ref ip) = device.manage
                                        if let Some(ref partial) = part
                                            ip.contains(partial)
                                        } else {
                                            true // Include all devices
                                        }
                                    } else {
                                        false
                                    }
                                })
                                .collect()
                        }
                        DeviceListFilter::Wlc { partial_wlc } => {
                            devices
                                .into_iter()
                                .filter(|device| {
                                    if let Some(ref wlc_ip) = device.as
                                        if let Some(ref partial) = part
                                            wlc_ip.contains(partial)
```

```
                                } else {
                                    true // Include all devices
                                }
                            } else {
                                false
                            }
                        })
                        .collect()
                }
            };

            utils::print_devices(filtered_devices);
        }
        Err(e) => error!("Failed to retrieve devices: {}", e),
    }
}
DeviceCommands::Detail { filter } => {
    // Fetch all devices
    match getdevicelist::get_all_devices(&config, &token).await
        Ok(devices) => {
            // Find the device matching the filter
            let device_option = match filter {
                DeviceDetailFilter::Hostname { ref hostname } =>
                    .into_iter()
                    .find(|device| device.hostname.as_deref() =
                DeviceDetailFilter::Mac { ref mac_address } =>
                    .into_iter()
                    .find(|device| device.mac_address.as_deref(
                DeviceDetailFilter::Ip { ref ip_address } => de
                    .into_iter()
                    .find(|device| {
                        device.management_ip_address.as_deref()
                    }),
            };

            match device_option {
                Some(device) => utils::print_device_detail(devi
                None => println!("No device found matching the
            }
        }
        Err(e) => error!("Failed to retrieve devices: {}", e),
    }
}
DeviceCommands::Enrichment { filter } => {
    // Handle the Enrichment command
    match filter {
        DeviceEnrichmentFilter::Mac { mac_address } => {
            match devicedetailenrichment::get_device_enrichment
                &config,
                &token,
                "mac_address",
                &mac_address,
```

```
                        )
                        .await
                        {
                            Ok(device_details) => {
                                utils::print_device_enrichment(device_detai
                            }
                            Err(e) => error!("Failed to retrieve device enr
                        }
                    }
                    DeviceEnrichmentFilter::Ip { ip_address } => {
                        match devicedetailenrichment::get_device_enrichment
                            &config,
                            &token,
                            "ip_address",
                            &ip_address,
                        )
                        .await
                        {
                            Ok(device_details) => {
                                utils::print_device_enrichment(device_detai
                            }
                            Err(e) => error!("Failed to retrieve device enr
                        }
                    }
                }
            }
        }
    });
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/show/client.rs

```
// src/handlers/show/client.rs

use crate::commands::show::client::ClientCommands;
use crate::app::config;
use crate::api::authentication::auth;
use crate::api::clients::{getclientdetail, getclientenrichment};
use crate::helpers::utils;
use log::error;

pub fn handle_client_command(subcommand: ClientCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        // Load configuration
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
```

```
                return;
            }
        };

        // Authenticate and get token
        let token = match auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
                error!("Authentication failed: {}", e);
                return;
            }
        };

        match subcommand {
            ClientCommands::Detail { mac_address } => {
                // Fetch client details
                match getclientdetail::get_client_detail(&config, &token, &
                    Ok(client_detail_response) => {
                        utils::print_client_detail(client_detail_response);
                    }
                    Err(e) => {
                        error!("Failed to retrieve client details: {}", e);
                    }
                }
            }
            ClientCommands::Enrichment {
                entity_type,
                entity_value,
                issue_category,
            } => {
                // Fetch client enrichment details
                match getclientenrichment::get_client_enrichment(
                    &config,
                    &token,
                    &entity_type,
                    &entity_value,
                    issue_category.as_deref(),
                )
                .await
                {
                    Ok(enrichment_response) => {
                        utils::print_client_enrichment(enrichment_response)
                    }
                    Err(e) => {
                        error!("Failed to retrieve client enrichment detail
                    }
                }
            }
        }
    });
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/show/mod.rs

```
pub mod device;
pub mod client;
pub mod issue;
pub mod ap;
use crate::commands::show::ShowCommands;

pub fn handle_show_command(subcommand: ShowCommands) {
    match subcommand {
        ShowCommands::Device { subcommand } => device::handle_device_comman

        ShowCommands::Client { subcommand } => client::handle_client_comman

        ShowCommands::Issue { subcommand } => issue::handle_issue_command(s

        ShowCommands::Ap { subcommand } => ap::handle_ap_command(subcommand
    }
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/show/issue.rs

```
// src/handlers/show/issue.rs

use crate::commands::show::issue::{IssueCommands, SearchOption};
use crate::app::config;
use crate::api::authentication::auth;
use crate::api::issues::getissuelist;
use crate::helpers::utils;
use log::error;
use std::collections::HashMap;

pub fn handle_issue_command(subcommand: IssueCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        // Load configuration
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        // Authenticate and get token
        let token = match auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
```

```
            error!("Authentication failed: {}", e);
            return;
        }
    };

    match subcommand {
        IssueCommands::List { search_option, search_input } => {
            // Prepare search parameters
            let mut search_params = HashMap::new();

            if let Some(option) = search_option {
                if let Some(input) = search_input {
                    match option {
                        SearchOption::StartTime => {
                            search_params.insert("startTime".to_string(
                        }
                        SearchOption::EndTime => {
                            search_params.insert("endTime".to_string(),
                        }
                        SearchOption::SiteId => {
                            search_params.insert("siteId".to_string(),
                        }
                        SearchOption::DeviceId => {
                            search_params.insert("deviceId".to_string()
                        }
                        SearchOption::MacAddress => {
                            search_params.insert("macAddress".to_string
                        }
                        SearchOption::Priority => {
                            search_params.insert("priority".to_string()
                        }
                        SearchOption::AiDriven => {
                            search_params.insert("aiDriven".to_string()
                        }
                        SearchOption::IssueStatus => {
                            search_params.insert("issueStatus".to_strin
                        }
                    }
                } else {
                    error!("Search input is required when a search opti
                    return;
                }
            }

            // Fetch issue list
            match getissuelist::get_issue_list(&config, &token, &search
                Ok(issue_list_response) => {
                    utils::print_issue_list(issue_list_response);
                }
                Err(e) => {
                    error!("Failed to retrieve issue list: {}", e);
                }
```

```
                }
            }
        }
    });
}
```

File: ./target/package/catalysh-0.0.2/src/handlers/mod.rs

```
pub mod show;
pub mod config;
pub mod app;

pub use show::handle_show_command;
pub use config::handle_config_command;
pub use app::handle_app_command;
```

File: ./target/package/catalysh-0.0.2/src/helpers/mod.rs

```
pub mod utils;
```

File: ./target/package/catalysh-0.0.2/src/helpers/utils.rs

```
// src/helpers/utils.rs
#[allow(unused_imports)]
use crate::api::clients::getclientdetail::{
    ClientDetailResponse,
    HealthScore as ClientDetailHealthScore,
    ConnectedDevice as ClientDetailConnectedDevice,
    Onboarding,
    ConnectionInfo,
    Topology,
    TopologyNode as ClientDetailTopologyNode,
    TopologyLink as ClientDetailTopologyLink,
};
#[allow(unused_imports)]
use crate::api::clients::getclientenrichment::{
    ClientEnrichmentResponse,
    ClientEnrichment,
    UserDetails,
    HealthScore as ClientEnrichmentHealthScore,
    ConnectedDevice as ClientEnrichmentConnectedDevice,
    DeviceDetails as ClientEnrichmentDeviceDetails,
    NeighborTopology,
    TopologyNode as ClientEnrichmentTopologyNode,
```

```
    TopologyLink as ClientEnrichmentTopologyLink,
    IssueDetails,
    Issue as ClientEnrichmentIssue,
    SuggestedAction,
    ImpactedHost,
    ImpactedHostLocation,
    VlanId,
};

#[allow(unused_imports)]
use crate::api::devices::devicedetailenrichment::DeviceDetails as DeviceDet
use crate::api::devices::getdevicelist::AllDevices;
use crate::api::clients::getclientenrichment::StringOrNumber;

#[allow(unused_imports)]
use crate::api::issues::getissuelist::{IssueListResponse, Issue as IssueLis
use crate::api::devices::devicedetailenrichment::DeviceDetails;
use crate::api::wireless::accesspointconfig::ApConfig;

use chrono::{DateTime, Utc};
use prettytable::{row, Table};

pub fn current_timestamp() -> u64 {
    Utc::now().timestamp_millis() as u64
}

// Function to print a list of devices
pub fn print_devices(devices: Vec<AllDevices>) {
    let mut table = Table::new();
    table.add_row(row![
        "Hostname",
        "Management IP",
        "Serial Number",
        "MAC Address",
        "Ethernet MAC Address",
        "Platform ID",
        "Software Version",
        "Role"
    ]);

    for device in devices {
        table.add_row(row![
            device.hostname.unwrap_or_else(|| "N/A".to_string()),
            device.management_ip_address.unwrap_or_else(|| "N/A".to_string(
            device.serial_number.unwrap_or_else(|| "N/A".to_string()),
            device.mac_address.unwrap_or_else(|| "N/A".to_string()),
            device.ap_ethernet_mac_address.unwrap_or_else(|| "N/A".to_strin
            device.platform_id.unwrap_or_else(|| "N/A".to_string()),
            device.software_version.unwrap_or_else(|| "N/A".to_string()),
            device.role.unwrap_or_else(|| "N/A".to_string()),
        ]);
    }
```

```
    table.printstd();
}

// Function to print detailed information about a device
pub fn print_device_detail(device: AllDevices) {
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Hostname", device.hostname);
    add_field(
        &mut table,
        "Management IP",
        device.management_ip_address,
    );
    add_field(&mut table, "Serial Number", device.serial_number);
    add_field(&mut table, "MAC Address", device.mac_address);
    add_field(&mut table, "Platform ID", device.platform_id);
    add_field(&mut table, "Software Version", device.software_version);
    add_field(&mut table, "Role", device.role);
    add_field(&mut table, "Reachability Status", device.reachability_status
    add_field(&mut table, "Uptime", device.up_time);
    add_field(&mut table, "Last Updated", device.last_update_time.map(|time
        let datetime = DateTime::from_timestamp_millis(timestamp as i64)
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expect("REASO
        datetime.format("%Y-%m-%d %H:%M:%S").to_string()
    }));
    // Add more fields as necessary

    table.printstd();
}

// Function to print enriched device details
pub fn print_device_enrichment(device_details: DeviceDetails) {
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Hostname", device_details.hostname);
    add_field(
        &mut table,
        "Management IP",
        device_details.managementIpAddress,
    );
    add_field(&mut table, "Serial Number", device_details.serialNumber);
    add_field(&mut table, "MAC Address", device_details.macAddress);
    add_field(
        &mut table,
        "Platform ID",
        device_details.platformId,
    );
    add_field(
        &mut table,
```

```
        "Software Version",
        device_details.softwareVersion,
    );
    add_field(
        &mut table,
        "Reachability Status",
        device_details.reachabilityStatus,
    );
    add_field(
        &mut table,
        "Error Code",
        device_details.errorCode.map(|v| v.to_string()),
    );
    add_field(
        &mut table,
        "Error Description",
        device_details.errorDescription,
    );
    // Add more fields as necessary

    table.printstd();
}

// Function to print client detail with all fields
pub fn print_client_detail(response: ClientDetailResponse) {
    if let Some(detail) = response.detail {
        let mut table = Table::new();
        table.add_row(row!["Field", "Value"]);

        add_field(&mut table, "ID", detail.id);
        add_field(&mut table, "Connection Status", detail.connectionStatus)
        add_field(&mut table, "Host Type", detail.hostType);
        add_field(&mut table, "User ID", detail.userId);
        add_field(&mut table, "Host Name", detail.hostName);
        add_field(&mut table, "Host OS", detail.hostOs);
        add_field(&mut table, "Host Version", detail.hostVersion);
        add_field(&mut table, "Sub Type", detail.subType);

        // lastUpdated as timestamp
        if let Some(timestamp) = detail.lastUpdated {
            let datetime = DateTime::from_timestamp_millis(timestamp as i64
                .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expect("R
            add_field(
                &mut table,
                "Last Updated",
                Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
            );
        } else {
            add_field(&mut table, "Last Updated", None);
        }

        // Health Score
```

```
if let Some(health_scores) = detail.healthScore {
    for (i, hs) in health_scores.iter().enumerate() {
        let prefix = format!("Health Score [{}]", i + 1);
        add_field(&mut table, &format!("{} - Health Type", prefix),
        add_field(&mut table, &format!("{} - Reason", prefix), hs.r
        add_field(
            &mut table,
            &format!("{} - Score", prefix),
            hs.score.map(|s| s.to_string()),
        );
    }
}

add_field(&mut table, "Host MAC", detail.hostMac);
add_field(&mut table, "Host IPv4", detail.hostIpV4);
add_field(
    &mut table,
    "Host IPv6",
    detail.hostIpV6.map(|ips| ips.join(", ")),
);
add_field(&mut table, "Auth Type", detail.authType);
add_field(
    &mut table,
    "VLAN ID",
    detail.vlanId.map(|v| v.to_string()),
);
add_field(
    &mut table,
    "VNID",
    detail.vnid.map(|v| v.to_string()),
);
add_field(&mut table, "SSID", detail.ssid);
add_field(&mut table, "Frequency", detail.frequency);
add_field(&mut table, "Channel", detail.channel);
add_field(&mut table, "AP Group", detail.apGroup);
add_field(&mut table, "Location", detail.location);
add_field(&mut table, "Client Connection", detail.clientConnection)

// Connected Devices
if let Some(connected_devices) = detail.connectedDevice {
    for (i, cd) in connected_devices.iter().enumerate() {
        let prefix = format!("Connected Device [{}]", i + 1);
        add_field(&mut table, &format!("{} - Type", prefix), cd.dev
        add_field(&mut table, &format!("{} - Name", prefix), cd.nam
        add_field(&mut table, &format!("{} - MAC", prefix), cd.mac.
        add_field(&mut table, &format!("{} - ID", prefix), cd.id.cl
        add_field(&mut table, &format!("{} - IP Address", prefix),
        add_field(&mut table, &format!("{} - Mgmt IP", prefix), cd.
        add_field(&mut table, &format!("{} - Band", prefix), cd.ban
        add_field(&mut table, &format!("{} - Mode", prefix), cd.mod
    }
}
```

```
add_field(
    &mut table,
    "Issue Count",
    detail.issueCount.map(|v| v.to_string()),
);
add_field(&mut table, "RSSI", detail.rssi);
add_field(&mut table, "Average RSSI", detail.avgRssi);
add_field(&mut table, "SNR", detail.snr);
add_field(&mut table, "Average SNR", detail.avgSnr);
add_field(&mut table, "Data Rate", detail.dataRate);
add_field(&mut table, "TX Bytes", detail.txBytes);
add_field(&mut table, "RX Bytes", detail.rxBytes);

// Onboarding
if let Some(onboarding) = detail.onboarding {
    // Timestamps
    if let Some(timestamp) = onboarding.authDoneTime {
        let datetime = DateTime::from_timestamp_millis(timestamp as
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expec
        add_field(
            &mut table,
            "Onboarding - Auth Done Time",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Onboarding - Auth Done Time", None);
    }

    if let Some(timestamp) = onboarding.assocDoneTime {
        let datetime = DateTime::from_timestamp_millis(timestamp as
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expec
        add_field(
            &mut table,
            "Onboarding - Assoc Done Time",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Onboarding - Assoc Done Time", None)
    }

    if let Some(timestamp) = onboarding.dhcpDoneTime {
        let datetime = DateTime::from_timestamp_millis(timestamp as
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expec
        add_field(
            &mut table,
            "Onboarding - DHCP Done Time",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Onboarding - DHCP Done Time", None);
    }
```

```
    // Other onboarding fields
    add_field(
        &mut table,
        "Onboarding - Average Run Duration",
        onboarding.averageRunDuration,
    );
    add_field(
        &mut table,
        "Onboarding - Max Run Duration",
        onboarding.maxRunDuration,
    );
    // Add more onboarding fields as necessary

    // Root cause lists
    if let Some(assoc_rc_list) = onboarding.assocRootcauseList {
        add_field(
            &mut table,
            "Onboarding - Assoc Rootcause List",
            Some(assoc_rc_list.join(", ")),
        );
    }
    if let Some(aaa_rc_list) = onboarding.aaaRootcauseList {
        add_field(
            &mut table,
            "Onboarding - AAA Rootcause List",
            Some(aaa_rc_list.join(", ")),
        );
    }
    if let Some(dhcp_rc_list) = onboarding.dhcpRootcauseList {
        add_field(
            &mut table,
            "Onboarding - DHCP Rootcause List",
            Some(dhcp_rc_list.join(", ")),
        );
    }
    if let Some(other_rc_list) = onboarding.otherRootcauseList {
        add_field(
            &mut table,
            "Onboarding - Other Rootcause List",
            Some(other_rc_list.join(", ")),
        );
    }
    if let Some(latest_rc_list) = onboarding.latestRootCauseList {
        add_field(
            &mut table,
            "Onboarding - Latest Rootcause List",
            Some(latest_rc_list.join(", ")),
        );
    }
}
```

```
    // Continue adding all other fields as needed

    table.printstd();
} else {
    println!("No client details available.");
}

// Optionally, print ConnectionInfo and Topology
if let Some(connection_info) = response.connectionInfo {
    println!("\nConnection Info:");
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Host Type", connection_info.hostType);
    add_field(&mut table, "Network Device Name", connection_info.nwDevi
    add_field(&mut table, "Network Device MAC", connection_info.nwDevic
    add_field(&mut table, "Protocol", connection_info.protocol);
    add_field(&mut table, "Band", connection_info.band);
    add_field(&mut table, "Spatial Stream", connection_info.spatialStre
    add_field(&mut table, "Channel", connection_info.channel);
    add_field(&mut table, "Channel Width", connection_info.channelWidth
    add_field(&mut table, "WMM", connection_info.wmm);
    add_field(&mut table, "UAPSD", connection_info.uapsd);

    // Timestamp
    if let Some(timestamp) = connection_info.timestamp {
        let datetime = DateTime::from_timestamp_millis(timestamp as i64
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expect("R
        add_field(
            &mut table,
            "Timestamp",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Timestamp", None);
    }

    table.printstd();
}

if let Some(topology) = response.topology {
    println!("\nTopology Information:");
    // You can choose to display topology data as needed
    if let Some(nodes) = topology.nodes {
        for node in nodes {
            let mut table = Table::new();
            table.add_row(row!["Node Field", "Value"]);
            add_field(&mut table, "Role", node.role);
            add_field(&mut table, "Name", node.name);
            add_field(&mut table, "ID", node.id);
            add_field(&mut table, "Description", node.description);
            add_field(&mut table, "Device Type", node.deviceType);
```

```
                add_field(&mut table, "Platform ID", node.platformId);
                add_field(&mut table, "Family", node.family);
                add_field(&mut table, "IP", node.ip);
                add_field(&mut table, "Software Version", node.softwareVers
                add_field(&mut table, "User ID", node.userId);
                add_field(&mut table, "Node Type", node.nodeType);
                add_field(&mut table, "Radio Frequency", node.radioFrequenc
                add_field(
                    &mut table,
                    "Clients",
                    node.clients.map(|v| v.to_string()),
                );
                add_field(
                    &mut table,
                    "Count",
                    node.count.map(|v| v.to_string()),
                );
                add_field(
                    &mut table,
                    "Health Score",
                    node.healthScore.map(|v| v.to_string()),
                );
                add_field(
                    &mut table,
                    "Level",
                    node.level.map(|v| v.to_string()),
                );
                add_field(&mut table, "Fabric Group", node.fabricGroup);
                add_field(&mut table, "Connected Device", node.connectedDev
                if let Some(fabric_roles) = node.fabricRole {
                    add_field(
                        &mut table,
                        "Fabric Roles",
                        Some(fabric_roles.join(", ")),
                    );
                }
                if let Some(ipv6_list) = node.ipv6 {
                    add_field(&mut table, "IPv6", Some(ipv6_list.join(", ")
                }

                table.printstd();
            }
        }

        // Similarly, you can display links if needed
    }
}

// Helper function to add a field to the table
fn add_field(table: &mut Table, field_name: &str, value: Option<String>) {
    table.add_row(row![
        field_name,
```

```
        value.unwrap_or_else(|| "N/A".to_string())
    ]);
}

pub fn print_issue_list(response: IssueListResponse) {
    if let Some(issues) = response.response {
        let mut table = Table::new();
        table.add_row(row![
            "Issue ID",
            "Name",
            "Device ID",
            "Device Role",
            "Client MAC",
            "Status",
            "Priority",
            "Category",
            "Last Occurrence Time"
        ]);

        for issue in issues {
            let last_occurrence = issue.last_occurence_time.map_or("N/A".to
                DateTime::from_timestamp_millis(timestamp)
                    .map(|dt| dt.format("%Y-%m-%d %H:%M:%S").to_string())
                    .unwrap_or_else(|| "Invalid Timestamp".to_string())
            });

            table.add_row(row![
                issue.issueId.clone().unwrap_or_else(|| "N/A".to_string()),
                issue.name.clone().unwrap_or_else(|| "N/A".to_string()),
                issue.deviceId.clone().unwrap_or_else(|| "N/A".to_string())
                issue.deviceRole.clone().unwrap_or_else(|| "N/A".to_string(
                issue.clientMac.clone().unwrap_or_else(|| "N/A".to_string()
                issue.status.clone().unwrap_or_else(|| "N/A".to_string()),
                issue.priority.clone().unwrap_or_else(|| "N/A".to_string())
                issue.category.clone().unwrap_or_else(|| "N/A".to_string())
                last_occurrence,
            ]);
        }

        table.printstd();
    } else {
        println!("No issues found.");
    }
}

// Function to print AP configuration
pub fn print_ap_config(ap_config: ApConfig) {
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Instance UUID", ap_config.instanceUuid.map(|v| v
    add_field(&mut table, "Instance ID", ap_config.instanceId.map(|v| v.to_
```

```
add_field(&mut table, "Display Name", ap_config.displayName);
add_field(&mut table, "Instance Tenant ID", ap_config.instanceTenantId)
add_field(
    &mut table,
    "Ordered List OE Index",
    ap_config._orderedListOEIndex.map(|v| v.to_string()),
);
add_field(
    &mut table,
    "Creation Order Index",
    ap_config._creationOrderIndex.map(|v| v.to_string()),
);
add_field(
    &mut table,
    "Is Being Changed",
    ap_config._isBeingChanged.map(|v| v.to_string()),
);
add_field(&mut table, "Deploy Pending", ap_config.deployPending);
add_field(&mut table, "Instance Version", ap_config.instanceVersion.map
add_field(&mut table, "Admin Status", ap_config.adminStatus);
add_field(&mut table, "AP Height", ap_config.apHeight.map(|v| v.to_stri
add_field(&mut table, "AP Mode", ap_config.apMode);
add_field(&mut table, "AP Name", ap_config.apName);
add_field(&mut table, "Ethernet MAC", ap_config.ethMac);
add_field(&mut table, "Failover Priority", ap_config.failoverPriority);
add_field(
    &mut table,
    "LED Brightness Level",
    ap_config.ledBrightnessLevel.map(|v| v.to_string()),
);
add_field(&mut table, "LED Status", ap_config.ledStatus);
add_field(&mut table, "Location", ap_config.location);
add_field(&mut table, "MAC Address", ap_config.macAddress);
add_field(&mut table, "Primary Controller Name", ap_config.primaryContr
add_field(&mut table, "Primary IP Address", ap_config.primaryIpAddress)
add_field(&mut table, "Secondary Controller Name", ap_config.secondaryC
add_field(&mut table, "Secondary IP Address", ap_config.secondaryIpAddr
add_field(&mut table, "Tertiary Controller Name", ap_config.tertiaryCon
add_field(&mut table, "Tertiary IP Address", ap_config.tertiaryIpAddres

// Internal Key
if let Some(internal_key) = ap_config.internalKey {
    add_field(&mut table, "Internal Key - Type", internal_key.type_fiel
    add_field(&mut table, "Internal Key - ID", internal_key.id.map(|v|
    add_field(&mut table, "Internal Key - Long Type", internal_key.long
    add_field(&mut table, "Internal Key - URL", internal_key.url);
}

// Display the table
table.printstd();

// Mesh DTOs - Since the schema shows as an array of empty objects, we
```

```
// Radio DTOs
if let Some(radio_dtos) = ap_config.radioDTOs {
    for (i, radio) in radio_dtos.iter().enumerate() {
        println!("\nRadio DTO [{}]:", i + 1);
        let mut radio_table = Table::new();
        radio_table.add_row(row!["Field", "Value"]);

        add_field(&mut radio_table, "Display Name", radio.displayName.c
        add_field(&mut radio_table, "Instance ID", radio.instanceId.map
        add_field(
            &mut radio_table,
            "Ordered List OE Index",
            radio._orderedListOEIndex.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Creation Order Index",
            radio._creationOrderIndex.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Is Being Changed",
            radio._isBeingChanged.map(|v| v.to_string()),
        );
        add_field(&mut radio_table, "Deploy Pending", radio.deployPendi
        add_field(
            &mut radio_table,
            "Instance Version",
            radio.instanceVersion.map(|v| v.to_string()),
        );
        add_field(&mut radio_table, "Admin Status", radio.adminStatus.c
        add_field(
            &mut radio_table,
            "Antenna Angle",
            radio.antennaAngle.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Antenna Elevation Angle",
            radio.antennaElevAngle.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Antenna Gain",
            radio.antennaGain.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Antenna Pattern Name",
            radio.antennaPatternName.clone(),
        );
```

```
add_field(
    &mut radio_table,
    "Channel Assignment Mode",
    radio.channelAssignmentMode.clone(),
);
add_field(
    &mut radio_table,
    "Channel Number",
    radio.channelNumber.map(|v| v.to_string()),
);
add_field(
    &mut radio_table,
    "Channel Width",
    radio.channelWidth.clone(),
);
add_field(&mut radio_table, "Clean Air SI", radio.cleanAirSI.cl
add_field(&mut radio_table, "Interface Type", radio.ifType.map(
add_field(
    &mut radio_table,
    "Interface Type Value",
    radio.ifTypeValue.clone(),
);
add_field(&mut radio_table, "MAC Address", radio.macAddress.clo
add_field(
    &mut radio_table,
    "Power Assignment Mode",
    radio.powerAssignmentMode.clone(),
);
add_field(
    &mut radio_table,
    "Power Level",
    radio.powerlevel.map(|v| v.to_string()),
);
// radioBand and radioRoleAssignment are Option<serde_json::Val
add_field(
    &mut radio_table,
    "Radio Band",
    radio.radioBand.as_ref().map(|v| v.to_string()),
);
add_field(
    &mut radio_table,
    "Radio Role Assignment",
    radio.radioRoleAssignment.as_ref().map(|v| v.to_string()),
);
add_field(&mut radio_table, "Slot ID", radio.slotId.map(|v| v.t

// Internal Key for RadioDTO
if let Some(radio_internal_key) = &radio.internalKey {
    add_field(
        &mut radio_table,
        "Internal Key - Type",
        radio_internal_key.type_field.clone(),
```

```
                );
                add_field(
                    &mut radio_table,
                    "Internal Key - ID",
                    radio_internal_key.id.map(|v| v.to_string()),
                );
                add_field(
                    &mut radio_table,
                    "Internal Key - Long Type",
                    radio_internal_key.longType.clone(),
                );
                add_field(
                    &mut radio_table,
                    "Internal Key - URL",
                    radio_internal_key.url.clone(),
                );
            }

            // Display the radio table
            radio_table.printstd();
        }
    }
}



pub fn print_client_enrichment(response: ClientEnrichmentResponse) {
    println!("Number of enrichment records: {}", response.0.len());

    for enrichment in response.0 {
        // User Details
        if let Some(user_details) = enrichment.userDetails {
            println!("User Details:");
            let mut table = Table::new();
            table.add_row(row!["Field", "Value"]);

            add_field(&mut table, "ID", user_details.id);
            add_field(&mut table, "Connection Status", user_details.connect
            add_field(&mut table, "Host Type", user_details.hostType);
            add_field(&mut table, "User ID", user_details.userId);
            add_field(&mut table, "Host Name", user_details.hostName);
            add_field(&mut table, "Host OS", user_details.hostOs);
            add_field(&mut table, "Host Version", user_details.hostVersion)
            add_field(&mut table, "Sub Type", user_details.subType);

            if let Some(timestamp) = user_details.lastUpdated {
                if let Some(datetime) = DateTime::from_timestamp_millis(tim
                    add_field(
                        &mut table,
                        "Last Updated",
                        Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string
                    );
```

```
    } else {
        add_field(&mut table, "Last Updated", Some("Invalid Tim
    }
} else {
    add_field(&mut table, "Last Updated", None);
}

// Health Scores
if let Some(health_scores) = user_details.healthScore {
    for (i, hs) in health_scores.iter().enumerate() {
        let prefix = format!("Health Score [{}]", i + 1);
        add_field(&mut table, &format!("{} - Health Type", pref
        add_field(&mut table, &format!("{} - Reason", prefix),
        add_field(
            &mut table,
            &format!("{} - Score", prefix),
            hs.score.map(|s| s.to_string()),
        );
    }
}

add_field(&mut table, "Host MAC", user_details.hostMac);
add_field(&mut table, "Host IPv4", user_details.hostIpV4);
add_field(
    &mut table,
    "Host IPv6",
    user_details.hostIpV6.map(|ips| ips.join(", ")),
);
add_field(&mut table, "Auth Type", user_details.authType);

// Handling vlanId that can be a string or a number
match user_details.vlanId {
    Some(VlanId::String(ref vlan)) => {
        add_field(&mut table, "VLAN ID", Some(vlan.clone()));
    }
    Some(VlanId::Number(vlan)) => {
        add_field(&mut table, "VLAN ID", Some(vlan.to_string())
    }
    None => {
        add_field(&mut table, "VLAN ID", None);
    }
}

add_field(&mut table, "SSID", user_details.ssid);
add_field(&mut table, "Location", user_details.location);
add_field(&mut table, "Client Connection", user_details.clientC

// Handling issueCount that can be a string or a number
match user_details.issueCount {
    Some(StringOrNumber::String(ref count)) => {
        add_field(&mut table, "Issue Count", Some(count.clone()
    }
```

```
    Some(StringOrNumber::Number(count)) => {
        add_field(&mut table, "Issue Count", Some(count.to_stri
    }
    None => {
        add_field(&mut table, "Issue Count", None);
    }
}

// Handling RSSI
match user_details.rssi {
    Some(StringOrNumber::String(ref value)) => {
        add_field(&mut table, "RSSI", Some(value.clone()));
    }
    Some(StringOrNumber::Number(value)) => {
        add_field(&mut table, "RSSI", Some(value.to_string()));
    }
    None => {
        add_field(&mut table, "RSSI", None);
    }
}

// Handling SNR
match user_details.snr {
    Some(StringOrNumber::String(ref value)) => {
        add_field(&mut table, "SNR", Some(value.clone()));
    }
    Some(StringOrNumber::Number(value)) => {
        add_field(&mut table, "SNR", Some(value.to_string()));
    }
    None => {
        add_field(&mut table, "SNR", None);
    }
}

// Handling Data Rate
match user_details.dataRate {
    Some(StringOrNumber::String(ref value)) => {
        add_field(&mut table, "Data Rate", Some(value.clone()))
    }
    Some(StringOrNumber::Number(value)) => {
        add_field(&mut table, "Data Rate", Some(value.to_string
    }
    None => {
        add_field(&mut table, "Data Rate", None);
    }
}

add_field(&mut table, "Port", user_details.port);

table.printstd();

// Onboarding Details
```

```
if let Some(onboarding) = user_details.onboarding {
    println!("Onboarding Details:");
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Average Run Duration", onboarding.av
    add_field(&mut table, "Max Run Duration", onboarding.maxRun
    add_field(&mut table, "Average Assoc Duration", onboarding.
    add_field(&mut table, "Max Assoc Duration", onboarding.maxA
    add_field(&mut table, "Average Auth Duration", onboarding.a
    add_field(&mut table, "Max Auth Duration", onboarding.maxAu
    add_field(&mut table, "Average DHCP Duration", onboarding.a
    add_field(&mut table, "Max DHCP Duration", onboarding.maxDh
    add_field(&mut table, "AAA Server IP", onboarding.aaaServer
    add_field(&mut table, "DHCP Server IP", onboarding.dhcpServ

    // Convert timestamps
    if let Some(auth_done_time) = onboarding.authDoneTime {
        if let Some(datetime) = DateTime::from_timestamp_millis
            add_field(
                &mut table,
                "Auth Done Time",
                Some(datetime.format("%Y-%m-%d %H:%M:%S").to_st
            );
        } else {
            add_field(&mut table, "Auth Done Time", Some("Inval
        }
    }

    if let Some(assoc_done_time) = onboarding.assocDoneTime {
        if let Some(datetime) = DateTime::from_timestamp_millis
            add_field(
                &mut table,
                "Assoc Done Time",
                Some(datetime.format("%Y-%m-%d %H:%M:%S").to_st
            );
        } else {
            add_field(&mut table, "Assoc Done Time", Some("Inva
        }
    }

    if let Some(dhcp_done_time) = onboarding.dhcpDoneTime {
        if let Some(datetime) = DateTime::from_timestamp_millis
            add_field(
                &mut table,
                "DHCP Done Time",
                Some(datetime.format("%Y-%m-%d %H:%M:%S").to_st
            );
        } else {
            add_field(&mut table, "DHCP Done Time", Some("Inval
        }
    }
```

```
            // Handle `latestRootCauseList`
            if let Some(root_causes) = onboarding.latestRootCauseList {
                add_field(&mut table, "Latest Root Causes", Some(root_c
            }

            table.printstd();
        }

        // Connected Devices in UserDetails
        if let Some(connected_devices) = user_details.connectedDevice {
            for (i, conn_dev) in connected_devices.iter().enumerate() {
                println!("\nConnected Device [{}]:", i + 1);
                let mut table = Table::new();
                table.add_row(row!["Field", "Value"]);

                add_field(&mut table, "Type", conn_dev.type_field.clone
                add_field(&mut table, "Name", conn_dev.name.clone());
                add_field(&mut table, "MAC", conn_dev.mac.clone());
                add_field(&mut table, "ID", conn_dev.id.clone());
                add_field(&mut table, "IP Address", conn_dev.ip_address
                add_field(&mut table, "Mgmt IP", conn_dev.mgmtIp.clone(
                add_field(&mut table, "Band", conn_dev.band.clone());
                add_field(&mut table, "Mode", conn_dev.mode.clone());

                table.printstd();
            }
        }
    } else {
        println!("User Details Missing");
    }

    // Connected Devices in Enrichment
    if let Some(connected_devices) = enrichment.connectedDevice {
        for (i, conn_dev) in connected_devices.iter().enumerate() {
            if let Some(device_details) = &conn_dev.deviceDetails {
                println!("\nConnected Device [{}]:", i + 1);
                let mut table = Table::new();
                table.add_row(row!["Field", "Value"]);

                add_field(&mut table, "Family", device_details.family.c
                add_field(&mut table, "Type", device_details.type_field
                add_field(&mut table, "Location", device_details.locati
                add_field(&mut table, "Error Code", device_details.erro
                add_field(&mut table, "MAC Address", device_details.mac
                add_field(&mut table, "Role", device_details.role.clone
                add_field(
                    &mut table,
                    "AP Manager Interface IP",
                    device_details.apManagerInterfaceIp.clone(),
                );
                add_field(
```

```
                    &mut table,
                    "Associated WLC IP",
                    device_details.associatedWlcIp.clone(),
                );
                add_field(&mut table, "Boot Date Time", device_details.
                add_field(
                    &mut table,
                    "Collection Status",
                    device_details.collectionStatus.clone(),
                );
                // Add more fields as needed

                table.printstd();
            }
        }
    } else {
        println!("Connected Devices Missing");
    }

    // Issue Details
    if let Some(issue_details) = enrichment.issueDetails {
        if let Some(issues) = issue_details.issue {
            for (i, issue) in issues.iter().enumerate() {
                println!("\nIssue [{}]:", i + 1);
                let mut table = Table::new();
                table.add_row(row!["Field", "Value"]);

                add_field(&mut table, "Issue ID", issue.issueId.clone()
                add_field(&mut table, "Issue Source", issue.issueSource
                add_field(&mut table, "Issue Category", issue.issueCate
                add_field(&mut table, "Issue Name", issue.issueName.clo
                add_field(&mut table, "Issue Description", issue.issueD
                add_field(&mut table, "Issue Entity", issue.issueEntity
                add_field(&mut table, "Issue Entity Value", issue.issue
                add_field(&mut table, "Issue Severity", issue.issueSeve
                add_field(&mut table, "Issue Priority", issue.issuePrio
                add_field(&mut table, "Issue Summary", issue.issueSumma
                if let Some(timestamp) = issue.issueTimestamp {
                    if let Some(datetime) = DateTime::from_timestamp_mi
                        add_field(
                            &mut table,
                            "Issue Timestamp",
                            Some(datetime.format("%Y-%m-%d %H:%M:%S").t
                        );
                    } else {
                        add_field(&mut table, "Issue Timestamp", Some("
                    }
                } else {
                    add_field(&mut table, "Issue Timestamp", None);
                }
                // Handle suggestedActions and impactedHosts if necessa
```

```
                    table.printstd();
                }
            } else {
                println!("No Issues Found");
            }
        } else {
            println!("Issue Details Missing");
        }
    }
}
```

File: ./target/debug/build/typenum-0568521468b798f9/out/consts.rs

```
/**
Type aliases for many constants.

This file is generated by typenum's build script.

For unsigned integers, the format is `U` followed by the number. We define

- Numbers 0 through 1024
- Powers of 2 below `u64::MAX`
- Powers of 10 below `u64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm};

# #[allow(dead_code)]
type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
```

For positive signed integers, the format is `P` followed by the number and
signed integers it is `N` followed by the number. For the signed integer ze
`Z0`. We define aliases for

- Numbers -1024 through 1024
- Powers of 2 between `i64::MIN` and `i64::MAX`
- Powers of 10 between `i64::MIN` and `i64::MAX`

These alias definitions look like this:

```rust
use typenum::{B0, B1, UInt, UTerm, PInt, NInt};

# #[allow(dead_code)]
type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
# #[allow(dead_code)]
```
```

```
type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

Example
```rust
# #[allow(unused_imports)]
use typenum::{U0, U1, U2, U3, U4, U5, U6};
# #[allow(unused_imports)]
use typenum::{N3, N2, N1, Z0, P1, P2, P3};
# #[allow(unused_imports)]
use typenum::{U774, N17, N10000, P1024, P4096};
```

We also define the aliases `False` and `True` for `B0` and `B1`, respective
*/
#[allow(missing_docs)]
pub mod consts {
    use crate::uint::{UInt, UTerm};
    use crate::int::{PInt, NInt};

    pub use crate::bit::{B0, B1};
    pub use crate::int::Z0;

    pub type True = B1;
    pub type False = B0;
    pub type U0 = UTerm;
    pub type U1 = UInt<UTerm, B1>;
    pub type P1 = PInt<U1>; pub type N1 = NInt<U1>;
    pub type U2 = UInt<UInt<UTerm, B1>, B0>;
    pub type P2 = PInt<U2>; pub type N2 = NInt<U2>;
    pub type U3 = UInt<UInt<UTerm, B1>, B1>;
    pub type P3 = PInt<U3>; pub type N3 = NInt<U3>;
    pub type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    pub type P4 = PInt<U4>; pub type N4 = NInt<U4>;
    pub type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    pub type P5 = PInt<U5>; pub type N5 = NInt<U5>;
    pub type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;
    pub type P6 = PInt<U6>; pub type N6 = NInt<U6>;
    pub type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
    pub type P7 = PInt<U7>; pub type N7 = NInt<U7>;
    pub type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
    pub type P8 = PInt<U8>; pub type N8 = NInt<U8>;
    pub type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;
    pub type P9 = PInt<U9>; pub type N9 = NInt<U9>;
    pub type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;
    pub type P10 = PInt<U10>; pub type N10 = NInt<U10>;
    pub type U11 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>;
    pub type P11 = PInt<U11>; pub type N11 = NInt<U11>;
    pub type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;
    pub type P12 = PInt<U12>; pub type N12 = NInt<U12>;
    pub type U13 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>;
    pub type P13 = PInt<U13>; pub type N13 = NInt<U13>;

```
pub type U14 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>;
pub type P14 = PInt<U14>; pub type N14 = NInt<U14>;
pub type U15 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;
pub type P15 = PInt<U15>; pub type N15 = NInt<U15>;
pub type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;
pub type P16 = PInt<U16>; pub type N16 = NInt<U16>;
pub type U17 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B1>;
pub type P17 = PInt<U17>; pub type N17 = NInt<U17>;
pub type U18 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B0>;
pub type P18 = PInt<U18>; pub type N18 = NInt<U18>;
pub type U19 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>, B1>;
pub type P19 = PInt<U19>; pub type N19 = NInt<U19>;
pub type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;
pub type P20 = PInt<U20>; pub type N20 = NInt<U20>;
pub type U21 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B1>;
pub type P21 = PInt<U21>; pub type N21 = NInt<U21>;
pub type U22 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B0>;
pub type P22 = PInt<U22>; pub type N22 = NInt<U22>;
pub type U23 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>, B1>;
pub type P23 = PInt<U23>; pub type N23 = NInt<U23>;
pub type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;
pub type P24 = PInt<U24>; pub type N24 = NInt<U24>;
pub type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
pub type P25 = PInt<U25>; pub type N25 = NInt<U25>;
pub type U26 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B0>;
pub type P26 = PInt<U26>; pub type N26 = NInt<U26>;
pub type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;
pub type P27 = PInt<U27>; pub type N27 = NInt<U27>;
pub type U28 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B0>;
pub type P28 = PInt<U28>; pub type N28 = NInt<U28>;
pub type U29 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>, B1>;
pub type P29 = PInt<U29>; pub type N29 = NInt<U29>;
pub type U30 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B0>;
pub type P30 = PInt<U30>; pub type N30 = NInt<U30>;
pub type U31 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>, B1>;
pub type P31 = PInt<U31>; pub type N31 = NInt<U31>;
pub type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P32 = PInt<U32>; pub type N32 = NInt<U32>;
pub type U33 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P33 = PInt<U33>; pub type N33 = NInt<U33>;
pub type U34 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P34 = PInt<U34>; pub type N34 = NInt<U34>;
pub type U35 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>,
pub type P35 = PInt<U35>; pub type N35 = NInt<U35>;
pub type U36 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P36 = PInt<U36>; pub type N36 = NInt<U36>;
pub type U37 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P37 = PInt<U37>; pub type N37 = NInt<U37>;
pub type U38 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P38 = PInt<U38>; pub type N38 = NInt<U38>;
pub type U39 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>,
pub type P39 = PInt<U39>; pub type N39 = NInt<U39>;
```

```
pub type U40 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P40 = PInt<U40>; pub type N40 = NInt<U40>;
pub type U41 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P41 = PInt<U41>; pub type N41 = NInt<U41>;
pub type U42 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P42 = PInt<U42>; pub type N42 = NInt<U42>;
pub type U43 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>,
pub type P43 = PInt<U43>; pub type N43 = NInt<U43>;
pub type U44 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P44 = PInt<U44>; pub type N44 = NInt<U44>;
pub type U45 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P45 = PInt<U45>; pub type N45 = NInt<U45>;
pub type U46 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P46 = PInt<U46>; pub type N46 = NInt<U46>;
pub type U47 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B1>,
pub type P47 = PInt<U47>; pub type N47 = NInt<U47>;
pub type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P48 = PInt<U48>; pub type N48 = NInt<U48>;
pub type U49 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P49 = PInt<U49>; pub type N49 = NInt<U49>;
pub type U50 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P50 = PInt<U50>; pub type N50 = NInt<U50>;
pub type U51 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>,
pub type P51 = PInt<U51>; pub type N51 = NInt<U51>;
pub type U52 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P52 = PInt<U52>; pub type N52 = NInt<U52>;
pub type U53 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P53 = PInt<U53>; pub type N53 = NInt<U53>;
pub type U54 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P54 = PInt<U54>; pub type N54 = NInt<U54>;
pub type U55 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>,
pub type P55 = PInt<U55>; pub type N55 = NInt<U55>;
pub type U56 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P56 = PInt<U56>; pub type N56 = NInt<U56>;
pub type U57 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P57 = PInt<U57>; pub type N57 = NInt<U57>;
pub type U58 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P58 = PInt<U58>; pub type N58 = NInt<U58>;
pub type U59 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B0>,
pub type P59 = PInt<U59>; pub type N59 = NInt<U59>;
pub type U60 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P60 = PInt<U60>; pub type N60 = NInt<U60>;
pub type U61 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P61 = PInt<U61>; pub type N61 = NInt<U61>;
pub type U62 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P62 = PInt<U62>; pub type N62 = NInt<U62>;
pub type U63 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>,
pub type P63 = PInt<U63>; pub type N63 = NInt<U63>;
pub type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P64 = PInt<U64>; pub type N64 = NInt<U64>;
pub type U65 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P65 = PInt<U65>; pub type N65 = NInt<U65>;
```

```
pub type U66 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P66 = PInt<U66>; pub type N66 = NInt<U66>;
pub type U67 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P67 = PInt<U67>; pub type N67 = NInt<U67>;
pub type U68 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P68 = PInt<U68>; pub type N68 = NInt<U68>;
pub type U69 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P69 = PInt<U69>; pub type N69 = NInt<U69>;
pub type U70 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P70 = PInt<U70>; pub type N70 = NInt<U70>;
pub type U71 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P71 = PInt<U71>; pub type N71 = NInt<U71>;
pub type U72 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P72 = PInt<U72>; pub type N72 = NInt<U72>;
pub type U73 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P73 = PInt<U73>; pub type N73 = NInt<U73>;
pub type U74 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P74 = PInt<U74>; pub type N74 = NInt<U74>;
pub type U75 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P75 = PInt<U75>; pub type N75 = NInt<U75>;
pub type U76 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P76 = PInt<U76>; pub type N76 = NInt<U76>;
pub type U77 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P77 = PInt<U77>; pub type N77 = NInt<U77>;
pub type U78 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P78 = PInt<U78>; pub type N78 = NInt<U78>;
pub type U79 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>,
pub type P79 = PInt<U79>; pub type N79 = NInt<U79>;
pub type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P80 = PInt<U80>; pub type N80 = NInt<U80>;
pub type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P81 = PInt<U81>; pub type N81 = NInt<U81>;
pub type U82 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P82 = PInt<U82>; pub type N82 = NInt<U82>;
pub type U83 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P83 = PInt<U83>; pub type N83 = NInt<U83>;
pub type U84 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P84 = PInt<U84>; pub type N84 = NInt<U84>;
pub type U85 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P85 = PInt<U85>; pub type N85 = NInt<U85>;
pub type U86 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P86 = PInt<U86>; pub type N86 = NInt<U86>;
pub type U87 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P87 = PInt<U87>; pub type N87 = NInt<U87>;
pub type U88 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P88 = PInt<U88>; pub type N88 = NInt<U88>;
pub type U89 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P89 = PInt<U89>; pub type N89 = NInt<U89>;
pub type U90 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P90 = PInt<U90>; pub type N90 = NInt<U90>;
pub type U91 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P91 = PInt<U91>; pub type N91 = NInt<U91>;
```

```
pub type U92 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P92 = PInt<U92>; pub type N92 = NInt<U92>;
pub type U93 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P93 = PInt<U93>; pub type N93 = NInt<U93>;
pub type U94 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P94 = PInt<U94>; pub type N94 = NInt<U94>;
pub type U95 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>,
pub type P95 = PInt<U95>; pub type N95 = NInt<U95>;
pub type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P96 = PInt<U96>; pub type N96 = NInt<U96>;
pub type U97 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P97 = PInt<U97>; pub type N97 = NInt<U97>;
pub type U98 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P98 = PInt<U98>; pub type N98 = NInt<U98>;
pub type U99 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>,
pub type P99 = PInt<U99>; pub type N99 = NInt<U99>;
pub type U100 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P100 = PInt<U100>; pub type N100 = NInt<U100>;
pub type U101 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P101 = PInt<U101>; pub type N101 = NInt<U101>;
pub type U102 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P102 = PInt<U102>; pub type N102 = NInt<U102>;
pub type U103 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P103 = PInt<U103>; pub type N103 = NInt<U103>;
pub type U104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P104 = PInt<U104>; pub type N104 = NInt<U104>;
pub type U105 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P105 = PInt<U105>; pub type N105 = NInt<U105>;
pub type U106 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P106 = PInt<U106>; pub type N106 = NInt<U106>;
pub type U107 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P107 = PInt<U107>; pub type N107 = NInt<U107>;
pub type U108 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P108 = PInt<U108>; pub type N108 = NInt<U108>;
pub type U109 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P109 = PInt<U109>; pub type N109 = NInt<U109>;
pub type U110 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P110 = PInt<U110>; pub type N110 = NInt<U110>;
pub type U111 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>
pub type P111 = PInt<U111>; pub type N111 = NInt<U111>;
pub type U112 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P112 = PInt<U112>; pub type N112 = NInt<U112>;
pub type U113 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P113 = PInt<U113>; pub type N113 = NInt<U113>;
pub type U114 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P114 = PInt<U114>; pub type N114 = NInt<U114>;
pub type U115 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P115 = PInt<U115>; pub type N115 = NInt<U115>;
pub type U116 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P116 = PInt<U116>; pub type N116 = NInt<U116>;
pub type U117 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P117 = PInt<U117>; pub type N117 = NInt<U117>;
```

```
pub type U118 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P118 = PInt<U118>; pub type N118 = NInt<U118>;
pub type U119 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P119 = PInt<U119>; pub type N119 = NInt<U119>;
pub type U120 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P120 = PInt<U120>; pub type N120 = NInt<U120>;
pub type U121 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P121 = PInt<U121>; pub type N121 = NInt<U121>;
pub type U122 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P122 = PInt<U122>; pub type N122 = NInt<U122>;
pub type U123 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P123 = PInt<U123>; pub type N123 = NInt<U123>;
pub type U124 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P124 = PInt<U124>; pub type N124 = NInt<U124>;
pub type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P125 = PInt<U125>; pub type N125 = NInt<U125>;
pub type U126 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P126 = PInt<U126>; pub type N126 = NInt<U126>;
pub type U127 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>
pub type P127 = PInt<U127>; pub type N127 = NInt<U127>;
pub type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P128 = PInt<U128>; pub type N128 = NInt<U128>;
pub type U129 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P129 = PInt<U129>; pub type N129 = NInt<U129>;
pub type U130 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P130 = PInt<U130>; pub type N130 = NInt<U130>;
pub type U131 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P131 = PInt<U131>; pub type N131 = NInt<U131>;
pub type U132 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P132 = PInt<U132>; pub type N132 = NInt<U132>;
pub type U133 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P133 = PInt<U133>; pub type N133 = NInt<U133>;
pub type U134 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P134 = PInt<U134>; pub type N134 = NInt<U134>;
pub type U135 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P135 = PInt<U135>; pub type N135 = NInt<U135>;
pub type U136 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P136 = PInt<U136>; pub type N136 = NInt<U136>;
pub type U137 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P137 = PInt<U137>; pub type N137 = NInt<U137>;
pub type U138 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P138 = PInt<U138>; pub type N138 = NInt<U138>;
pub type U139 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P139 = PInt<U139>; pub type N139 = NInt<U139>;
pub type U140 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P140 = PInt<U140>; pub type N140 = NInt<U140>;
pub type U141 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P141 = PInt<U141>; pub type N141 = NInt<U141>;
pub type U142 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P142 = PInt<U142>; pub type N142 = NInt<U142>;
pub type U143 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P143 = PInt<U143>; pub type N143 = NInt<U143>;
```

```
pub type U144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P144 = PInt<U144>; pub type N144 = NInt<U144>;
pub type U145 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P145 = PInt<U145>; pub type N145 = NInt<U145>;
pub type U146 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P146 = PInt<U146>; pub type N146 = NInt<U146>;
pub type U147 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P147 = PInt<U147>; pub type N147 = NInt<U147>;
pub type U148 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P148 = PInt<U148>; pub type N148 = NInt<U148>;
pub type U149 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P149 = PInt<U149>; pub type N149 = NInt<U149>;
pub type U150 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P150 = PInt<U150>; pub type N150 = NInt<U150>;
pub type U151 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P151 = PInt<U151>; pub type N151 = NInt<U151>;
pub type U152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P152 = PInt<U152>; pub type N152 = NInt<U152>;
pub type U153 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P153 = PInt<U153>; pub type N153 = NInt<U153>;
pub type U154 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P154 = PInt<U154>; pub type N154 = NInt<U154>;
pub type U155 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P155 = PInt<U155>; pub type N155 = NInt<U155>;
pub type U156 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P156 = PInt<U156>; pub type N156 = NInt<U156>;
pub type U157 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P157 = PInt<U157>; pub type N157 = NInt<U157>;
pub type U158 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P158 = PInt<U158>; pub type N158 = NInt<U158>;
pub type U159 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P159 = PInt<U159>; pub type N159 = NInt<U159>;
pub type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P160 = PInt<U160>; pub type N160 = NInt<U160>;
pub type U161 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P161 = PInt<U161>; pub type N161 = NInt<U161>;
pub type U162 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P162 = PInt<U162>; pub type N162 = NInt<U162>;
pub type U163 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P163 = PInt<U163>; pub type N163 = NInt<U163>;
pub type U164 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P164 = PInt<U164>; pub type N164 = NInt<U164>;
pub type U165 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P165 = PInt<U165>; pub type N165 = NInt<U165>;
pub type U166 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P166 = PInt<U166>; pub type N166 = NInt<U166>;
pub type U167 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P167 = PInt<U167>; pub type N167 = NInt<U167>;
pub type U168 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P168 = PInt<U168>; pub type N168 = NInt<U168>;
pub type U169 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P169 = PInt<U169>; pub type N169 = NInt<U169>;
```

```
pub type U170 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P170 = PInt<U170>; pub type N170 = NInt<U170>;
pub type U171 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P171 = PInt<U171>; pub type N171 = NInt<U171>;
pub type U172 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P172 = PInt<U172>; pub type N172 = NInt<U172>;
pub type U173 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P173 = PInt<U173>; pub type N173 = NInt<U173>;
pub type U174 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P174 = PInt<U174>; pub type N174 = NInt<U174>;
pub type U175 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P175 = PInt<U175>; pub type N175 = NInt<U175>;
pub type U176 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P176 = PInt<U176>; pub type N176 = NInt<U176>;
pub type U177 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P177 = PInt<U177>; pub type N177 = NInt<U177>;
pub type U178 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P178 = PInt<U178>; pub type N178 = NInt<U178>;
pub type U179 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P179 = PInt<U179>; pub type N179 = NInt<U179>;
pub type U180 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P180 = PInt<U180>; pub type N180 = NInt<U180>;
pub type U181 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P181 = PInt<U181>; pub type N181 = NInt<U181>;
pub type U182 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P182 = PInt<U182>; pub type N182 = NInt<U182>;
pub type U183 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P183 = PInt<U183>; pub type N183 = NInt<U183>;
pub type U184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P184 = PInt<U184>; pub type N184 = NInt<U184>;
pub type U185 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P185 = PInt<U185>; pub type N185 = NInt<U185>;
pub type U186 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P186 = PInt<U186>; pub type N186 = NInt<U186>;
pub type U187 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P187 = PInt<U187>; pub type N187 = NInt<U187>;
pub type U188 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P188 = PInt<U188>; pub type N188 = NInt<U188>;
pub type U189 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P189 = PInt<U189>; pub type N189 = NInt<U189>;
pub type U190 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P190 = PInt<U190>; pub type N190 = NInt<U190>;
pub type U191 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>
pub type P191 = PInt<U191>; pub type N191 = NInt<U191>;
pub type U192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P192 = PInt<U192>; pub type N192 = NInt<U192>;
pub type U193 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P193 = PInt<U193>; pub type N193 = NInt<U193>;
pub type U194 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P194 = PInt<U194>; pub type N194 = NInt<U194>;
pub type U195 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P195 = PInt<U195>; pub type N195 = NInt<U195>;
```

```
pub type U196 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P196 = PInt<U196>; pub type N196 = NInt<U196>;
pub type U197 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P197 = PInt<U197>; pub type N197 = NInt<U197>;
pub type U198 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P198 = PInt<U198>; pub type N198 = NInt<U198>;
pub type U199 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P199 = PInt<U199>; pub type N199 = NInt<U199>;
pub type U200 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P200 = PInt<U200>; pub type N200 = NInt<U200>;
pub type U201 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P201 = PInt<U201>; pub type N201 = NInt<U201>;
pub type U202 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P202 = PInt<U202>; pub type N202 = NInt<U202>;
pub type U203 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P203 = PInt<U203>; pub type N203 = NInt<U203>;
pub type U204 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P204 = PInt<U204>; pub type N204 = NInt<U204>;
pub type U205 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P205 = PInt<U205>; pub type N205 = NInt<U205>;
pub type U206 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P206 = PInt<U206>; pub type N206 = NInt<U206>;
pub type U207 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P207 = PInt<U207>; pub type N207 = NInt<U207>;
pub type U208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P208 = PInt<U208>; pub type N208 = NInt<U208>;
pub type U209 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P209 = PInt<U209>; pub type N209 = NInt<U209>;
pub type U210 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P210 = PInt<U210>; pub type N210 = NInt<U210>;
pub type U211 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P211 = PInt<U211>; pub type N211 = NInt<U211>;
pub type U212 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P212 = PInt<U212>; pub type N212 = NInt<U212>;
pub type U213 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P213 = PInt<U213>; pub type N213 = NInt<U213>;
pub type U214 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P214 = PInt<U214>; pub type N214 = NInt<U214>;
pub type U215 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P215 = PInt<U215>; pub type N215 = NInt<U215>;
pub type U216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P216 = PInt<U216>; pub type N216 = NInt<U216>;
pub type U217 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P217 = PInt<U217>; pub type N217 = NInt<U217>;
pub type U218 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P218 = PInt<U218>; pub type N218 = NInt<U218>;
pub type U219 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P219 = PInt<U219>; pub type N219 = NInt<U219>;
pub type U220 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P220 = PInt<U220>; pub type N220 = NInt<U220>;
pub type U221 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P221 = PInt<U221>; pub type N221 = NInt<U221>;
```

```
pub type U222 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P222 = PInt<U222>; pub type N222 = NInt<U222>;
pub type U223 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P223 = PInt<U223>; pub type N223 = NInt<U223>;
pub type U224 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P224 = PInt<U224>; pub type N224 = NInt<U224>;
pub type U225 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P225 = PInt<U225>; pub type N225 = NInt<U225>;
pub type U226 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P226 = PInt<U226>; pub type N226 = NInt<U226>;
pub type U227 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P227 = PInt<U227>; pub type N227 = NInt<U227>;
pub type U228 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P228 = PInt<U228>; pub type N228 = NInt<U228>;
pub type U229 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P229 = PInt<U229>; pub type N229 = NInt<U229>;
pub type U230 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P230 = PInt<U230>; pub type N230 = NInt<U230>;
pub type U231 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P231 = PInt<U231>; pub type N231 = NInt<U231>;
pub type U232 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P232 = PInt<U232>; pub type N232 = NInt<U232>;
pub type U233 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P233 = PInt<U233>; pub type N233 = NInt<U233>;
pub type U234 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P234 = PInt<U234>; pub type N234 = NInt<U234>;
pub type U235 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P235 = PInt<U235>; pub type N235 = NInt<U235>;
pub type U236 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P236 = PInt<U236>; pub type N236 = NInt<U236>;
pub type U237 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P237 = PInt<U237>; pub type N237 = NInt<U237>;
pub type U238 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P238 = PInt<U238>; pub type N238 = NInt<U238>;
pub type U239 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P239 = PInt<U239>; pub type N239 = NInt<U239>;
pub type U240 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P240 = PInt<U240>; pub type N240 = NInt<U240>;
pub type U241 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P241 = PInt<U241>; pub type N241 = NInt<U241>;
pub type U242 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P242 = PInt<U242>; pub type N242 = NInt<U242>;
pub type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P243 = PInt<U243>; pub type N243 = NInt<U243>;
pub type U244 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P244 = PInt<U244>; pub type N244 = NInt<U244>;
pub type U245 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P245 = PInt<U245>; pub type N245 = NInt<U245>;
pub type U246 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P246 = PInt<U246>; pub type N246 = NInt<U246>;
pub type U247 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P247 = PInt<U247>; pub type N247 = NInt<U247>;
```

```
pub type U248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P248 = PInt<U248>; pub type N248 = NInt<U248>;
pub type U249 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P249 = PInt<U249>; pub type N249 = NInt<U249>;
pub type U250 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P250 = PInt<U250>; pub type N250 = NInt<U250>;
pub type U251 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P251 = PInt<U251>; pub type N251 = NInt<U251>;
pub type U252 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P252 = PInt<U252>; pub type N252 = NInt<U252>;
pub type U253 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P253 = PInt<U253>; pub type N253 = NInt<U253>;
pub type U254 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P254 = PInt<U254>; pub type N254 = NInt<U254>;
pub type U255 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>
pub type P255 = PInt<U255>; pub type N255 = NInt<U255>;
pub type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P256 = PInt<U256>; pub type N256 = NInt<U256>;
pub type U257 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P257 = PInt<U257>; pub type N257 = NInt<U257>;
pub type U258 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P258 = PInt<U258>; pub type N258 = NInt<U258>;
pub type U259 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P259 = PInt<U259>; pub type N259 = NInt<U259>;
pub type U260 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P260 = PInt<U260>; pub type N260 = NInt<U260>;
pub type U261 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P261 = PInt<U261>; pub type N261 = NInt<U261>;
pub type U262 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P262 = PInt<U262>; pub type N262 = NInt<U262>;
pub type U263 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P263 = PInt<U263>; pub type N263 = NInt<U263>;
pub type U264 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P264 = PInt<U264>; pub type N264 = NInt<U264>;
pub type U265 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P265 = PInt<U265>; pub type N265 = NInt<U265>;
pub type U266 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P266 = PInt<U266>; pub type N266 = NInt<U266>;
pub type U267 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P267 = PInt<U267>; pub type N267 = NInt<U267>;
pub type U268 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P268 = PInt<U268>; pub type N268 = NInt<U268>;
pub type U269 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P269 = PInt<U269>; pub type N269 = NInt<U269>;
pub type U270 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P270 = PInt<U270>; pub type N270 = NInt<U270>;
pub type U271 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P271 = PInt<U271>; pub type N271 = NInt<U271>;
pub type U272 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P272 = PInt<U272>; pub type N272 = NInt<U272>;
pub type U273 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P273 = PInt<U273>; pub type N273 = NInt<U273>;
```

```
pub type U274 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P274 = PInt<U274>; pub type N274 = NInt<U274>;
pub type U275 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P275 = PInt<U275>; pub type N275 = NInt<U275>;
pub type U276 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P276 = PInt<U276>; pub type N276 = NInt<U276>;
pub type U277 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P277 = PInt<U277>; pub type N277 = NInt<U277>;
pub type U278 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P278 = PInt<U278>; pub type N278 = NInt<U278>;
pub type U279 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P279 = PInt<U279>; pub type N279 = NInt<U279>;
pub type U280 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P280 = PInt<U280>; pub type N280 = NInt<U280>;
pub type U281 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P281 = PInt<U281>; pub type N281 = NInt<U281>;
pub type U282 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P282 = PInt<U282>; pub type N282 = NInt<U282>;
pub type U283 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P283 = PInt<U283>; pub type N283 = NInt<U283>;
pub type U284 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P284 = PInt<U284>; pub type N284 = NInt<U284>;
pub type U285 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P285 = PInt<U285>; pub type N285 = NInt<U285>;
pub type U286 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P286 = PInt<U286>; pub type N286 = NInt<U286>;
pub type U287 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P287 = PInt<U287>; pub type N287 = NInt<U287>;
pub type U288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P288 = PInt<U288>; pub type N288 = NInt<U288>;
pub type U289 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P289 = PInt<U289>; pub type N289 = NInt<U289>;
pub type U290 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P290 = PInt<U290>; pub type N290 = NInt<U290>;
pub type U291 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P291 = PInt<U291>; pub type N291 = NInt<U291>;
pub type U292 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P292 = PInt<U292>; pub type N292 = NInt<U292>;
pub type U293 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P293 = PInt<U293>; pub type N293 = NInt<U293>;
pub type U294 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P294 = PInt<U294>; pub type N294 = NInt<U294>;
pub type U295 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P295 = PInt<U295>; pub type N295 = NInt<U295>;
pub type U296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P296 = PInt<U296>; pub type N296 = NInt<U296>;
pub type U297 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P297 = PInt<U297>; pub type N297 = NInt<U297>;
pub type U298 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P298 = PInt<U298>; pub type N298 = NInt<U298>;
pub type U299 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P299 = PInt<U299>; pub type N299 = NInt<U299>;
```

```
pub type U300 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P300 = PInt<U300>; pub type N300 = NInt<U300>;
pub type U301 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P301 = PInt<U301>; pub type N301 = NInt<U301>;
pub type U302 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P302 = PInt<U302>; pub type N302 = NInt<U302>;
pub type U303 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P303 = PInt<U303>; pub type N303 = NInt<U303>;
pub type U304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P304 = PInt<U304>; pub type N304 = NInt<U304>;
pub type U305 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P305 = PInt<U305>; pub type N305 = NInt<U305>;
pub type U306 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P306 = PInt<U306>; pub type N306 = NInt<U306>;
pub type U307 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P307 = PInt<U307>; pub type N307 = NInt<U307>;
pub type U308 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P308 = PInt<U308>; pub type N308 = NInt<U308>;
pub type U309 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P309 = PInt<U309>; pub type N309 = NInt<U309>;
pub type U310 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P310 = PInt<U310>; pub type N310 = NInt<U310>;
pub type U311 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P311 = PInt<U311>; pub type N311 = NInt<U311>;
pub type U312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P312 = PInt<U312>; pub type N312 = NInt<U312>;
pub type U313 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P313 = PInt<U313>; pub type N313 = NInt<U313>;
pub type U314 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P314 = PInt<U314>; pub type N314 = NInt<U314>;
pub type U315 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P315 = PInt<U315>; pub type N315 = NInt<U315>;
pub type U316 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P316 = PInt<U316>; pub type N316 = NInt<U316>;
pub type U317 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P317 = PInt<U317>; pub type N317 = NInt<U317>;
pub type U318 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P318 = PInt<U318>; pub type N318 = NInt<U318>;
pub type U319 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P319 = PInt<U319>; pub type N319 = NInt<U319>;
pub type U320 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P320 = PInt<U320>; pub type N320 = NInt<U320>;
pub type U321 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P321 = PInt<U321>; pub type N321 = NInt<U321>;
pub type U322 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P322 = PInt<U322>; pub type N322 = NInt<U322>;
pub type U323 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P323 = PInt<U323>; pub type N323 = NInt<U323>;
pub type U324 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P324 = PInt<U324>; pub type N324 = NInt<U324>;
pub type U325 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P325 = PInt<U325>; pub type N325 = NInt<U325>;
```

```
pub type U326 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P326 = PInt<U326>; pub type N326 = NInt<U326>;
pub type U327 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P327 = PInt<U327>; pub type N327 = NInt<U327>;
pub type U328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P328 = PInt<U328>; pub type N328 = NInt<U328>;
pub type U329 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P329 = PInt<U329>; pub type N329 = NInt<U329>;
pub type U330 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P330 = PInt<U330>; pub type N330 = NInt<U330>;
pub type U331 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P331 = PInt<U331>; pub type N331 = NInt<U331>;
pub type U332 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P332 = PInt<U332>; pub type N332 = NInt<U332>;
pub type U333 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P333 = PInt<U333>; pub type N333 = NInt<U333>;
pub type U334 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P334 = PInt<U334>; pub type N334 = NInt<U334>;
pub type U335 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P335 = PInt<U335>; pub type N335 = NInt<U335>;
pub type U336 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P336 = PInt<U336>; pub type N336 = NInt<U336>;
pub type U337 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P337 = PInt<U337>; pub type N337 = NInt<U337>;
pub type U338 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P338 = PInt<U338>; pub type N338 = NInt<U338>;
pub type U339 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P339 = PInt<U339>; pub type N339 = NInt<U339>;
pub type U340 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P340 = PInt<U340>; pub type N340 = NInt<U340>;
pub type U341 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P341 = PInt<U341>; pub type N341 = NInt<U341>;
pub type U342 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P342 = PInt<U342>; pub type N342 = NInt<U342>;
pub type U343 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P343 = PInt<U343>; pub type N343 = NInt<U343>;
pub type U344 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P344 = PInt<U344>; pub type N344 = NInt<U344>;
pub type U345 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P345 = PInt<U345>; pub type N345 = NInt<U345>;
pub type U346 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P346 = PInt<U346>; pub type N346 = NInt<U346>;
pub type U347 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P347 = PInt<U347>; pub type N347 = NInt<U347>;
pub type U348 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P348 = PInt<U348>; pub type N348 = NInt<U348>;
pub type U349 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P349 = PInt<U349>; pub type N349 = NInt<U349>;
pub type U350 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P350 = PInt<U350>; pub type N350 = NInt<U350>;
pub type U351 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P351 = PInt<U351>; pub type N351 = NInt<U351>;
```

```
pub type U352 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P352 = PInt<U352>; pub type N352 = NInt<U352>;
pub type U353 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P353 = PInt<U353>; pub type N353 = NInt<U353>;
pub type U354 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P354 = PInt<U354>; pub type N354 = NInt<U354>;
pub type U355 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P355 = PInt<U355>; pub type N355 = NInt<U355>;
pub type U356 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P356 = PInt<U356>; pub type N356 = NInt<U356>;
pub type U357 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P357 = PInt<U357>; pub type N357 = NInt<U357>;
pub type U358 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P358 = PInt<U358>; pub type N358 = NInt<U358>;
pub type U359 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P359 = PInt<U359>; pub type N359 = NInt<U359>;
pub type U360 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P360 = PInt<U360>; pub type N360 = NInt<U360>;
pub type U361 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P361 = PInt<U361>; pub type N361 = NInt<U361>;
pub type U362 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P362 = PInt<U362>; pub type N362 = NInt<U362>;
pub type U363 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P363 = PInt<U363>; pub type N363 = NInt<U363>;
pub type U364 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P364 = PInt<U364>; pub type N364 = NInt<U364>;
pub type U365 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P365 = PInt<U365>; pub type N365 = NInt<U365>;
pub type U366 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P366 = PInt<U366>; pub type N366 = NInt<U366>;
pub type U367 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P367 = PInt<U367>; pub type N367 = NInt<U367>;
pub type U368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P368 = PInt<U368>; pub type N368 = NInt<U368>;
pub type U369 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P369 = PInt<U369>; pub type N369 = NInt<U369>;
pub type U370 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P370 = PInt<U370>; pub type N370 = NInt<U370>;
pub type U371 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P371 = PInt<U371>; pub type N371 = NInt<U371>;
pub type U372 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P372 = PInt<U372>; pub type N372 = NInt<U372>;
pub type U373 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P373 = PInt<U373>; pub type N373 = NInt<U373>;
pub type U374 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P374 = PInt<U374>; pub type N374 = NInt<U374>;
pub type U375 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P375 = PInt<U375>; pub type N375 = NInt<U375>;
pub type U376 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P376 = PInt<U376>; pub type N376 = NInt<U376>;
pub type U377 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P377 = PInt<U377>; pub type N377 = NInt<U377>;
```

```
pub type U378 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P378 = PInt<U378>; pub type N378 = NInt<U378>;
pub type U379 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P379 = PInt<U379>; pub type N379 = NInt<U379>;
pub type U380 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P380 = PInt<U380>; pub type N380 = NInt<U380>;
pub type U381 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P381 = PInt<U381>; pub type N381 = NInt<U381>;
pub type U382 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P382 = PInt<U382>; pub type N382 = NInt<U382>;
pub type U383 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P383 = PInt<U383>; pub type N383 = NInt<U383>;
pub type U384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P384 = PInt<U384>; pub type N384 = NInt<U384>;
pub type U385 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P385 = PInt<U385>; pub type N385 = NInt<U385>;
pub type U386 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P386 = PInt<U386>; pub type N386 = NInt<U386>;
pub type U387 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P387 = PInt<U387>; pub type N387 = NInt<U387>;
pub type U388 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P388 = PInt<U388>; pub type N388 = NInt<U388>;
pub type U389 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P389 = PInt<U389>; pub type N389 = NInt<U389>;
pub type U390 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P390 = PInt<U390>; pub type N390 = NInt<U390>;
pub type U391 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P391 = PInt<U391>; pub type N391 = NInt<U391>;
pub type U392 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P392 = PInt<U392>; pub type N392 = NInt<U392>;
pub type U393 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P393 = PInt<U393>; pub type N393 = NInt<U393>;
pub type U394 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P394 = PInt<U394>; pub type N394 = NInt<U394>;
pub type U395 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P395 = PInt<U395>; pub type N395 = NInt<U395>;
pub type U396 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P396 = PInt<U396>; pub type N396 = NInt<U396>;
pub type U397 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P397 = PInt<U397>; pub type N397 = NInt<U397>;
pub type U398 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P398 = PInt<U398>; pub type N398 = NInt<U398>;
pub type U399 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P399 = PInt<U399>; pub type N399 = NInt<U399>;
pub type U400 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P400 = PInt<U400>; pub type N400 = NInt<U400>;
pub type U401 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P401 = PInt<U401>; pub type N401 = NInt<U401>;
pub type U402 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P402 = PInt<U402>; pub type N402 = NInt<U402>;
pub type U403 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P403 = PInt<U403>; pub type N403 = NInt<U403>;
```

```
pub type U404 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P404 = PInt<U404>; pub type N404 = NInt<U404>;
pub type U405 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P405 = PInt<U405>; pub type N405 = NInt<U405>;
pub type U406 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P406 = PInt<U406>; pub type N406 = NInt<U406>;
pub type U407 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P407 = PInt<U407>; pub type N407 = NInt<U407>;
pub type U408 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P408 = PInt<U408>; pub type N408 = NInt<U408>;
pub type U409 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P409 = PInt<U409>; pub type N409 = NInt<U409>;
pub type U410 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P410 = PInt<U410>; pub type N410 = NInt<U410>;
pub type U411 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P411 = PInt<U411>; pub type N411 = NInt<U411>;
pub type U412 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P412 = PInt<U412>; pub type N412 = NInt<U412>;
pub type U413 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P413 = PInt<U413>; pub type N413 = NInt<U413>;
pub type U414 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P414 = PInt<U414>; pub type N414 = NInt<U414>;
pub type U415 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P415 = PInt<U415>; pub type N415 = NInt<U415>;
pub type U416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P416 = PInt<U416>; pub type N416 = NInt<U416>;
pub type U417 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P417 = PInt<U417>; pub type N417 = NInt<U417>;
pub type U418 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P418 = PInt<U418>; pub type N418 = NInt<U418>;
pub type U419 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P419 = PInt<U419>; pub type N419 = NInt<U419>;
pub type U420 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P420 = PInt<U420>; pub type N420 = NInt<U420>;
pub type U421 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P421 = PInt<U421>; pub type N421 = NInt<U421>;
pub type U422 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P422 = PInt<U422>; pub type N422 = NInt<U422>;
pub type U423 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P423 = PInt<U423>; pub type N423 = NInt<U423>;
pub type U424 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P424 = PInt<U424>; pub type N424 = NInt<U424>;
pub type U425 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P425 = PInt<U425>; pub type N425 = NInt<U425>;
pub type U426 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P426 = PInt<U426>; pub type N426 = NInt<U426>;
pub type U427 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P427 = PInt<U427>; pub type N427 = NInt<U427>;
pub type U428 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P428 = PInt<U428>; pub type N428 = NInt<U428>;
pub type U429 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P429 = PInt<U429>; pub type N429 = NInt<U429>;
```

```
pub type U430 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P430 = PInt<U430>; pub type N430 = NInt<U430>;
pub type U431 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P431 = PInt<U431>; pub type N431 = NInt<U431>;
pub type U432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P432 = PInt<U432>; pub type N432 = NInt<U432>;
pub type U433 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P433 = PInt<U433>; pub type N433 = NInt<U433>;
pub type U434 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P434 = PInt<U434>; pub type N434 = NInt<U434>;
pub type U435 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P435 = PInt<U435>; pub type N435 = NInt<U435>;
pub type U436 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P436 = PInt<U436>; pub type N436 = NInt<U436>;
pub type U437 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P437 = PInt<U437>; pub type N437 = NInt<U437>;
pub type U438 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P438 = PInt<U438>; pub type N438 = NInt<U438>;
pub type U439 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P439 = PInt<U439>; pub type N439 = NInt<U439>;
pub type U440 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P440 = PInt<U440>; pub type N440 = NInt<U440>;
pub type U441 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P441 = PInt<U441>; pub type N441 = NInt<U441>;
pub type U442 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P442 = PInt<U442>; pub type N442 = NInt<U442>;
pub type U443 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P443 = PInt<U443>; pub type N443 = NInt<U443>;
pub type U444 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P444 = PInt<U444>; pub type N444 = NInt<U444>;
pub type U445 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P445 = PInt<U445>; pub type N445 = NInt<U445>;
pub type U446 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P446 = PInt<U446>; pub type N446 = NInt<U446>;
pub type U447 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P447 = PInt<U447>; pub type N447 = NInt<U447>;
pub type U448 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P448 = PInt<U448>; pub type N448 = NInt<U448>;
pub type U449 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P449 = PInt<U449>; pub type N449 = NInt<U449>;
pub type U450 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P450 = PInt<U450>; pub type N450 = NInt<U450>;
pub type U451 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P451 = PInt<U451>; pub type N451 = NInt<U451>;
pub type U452 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P452 = PInt<U452>; pub type N452 = NInt<U452>;
pub type U453 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P453 = PInt<U453>; pub type N453 = NInt<U453>;
pub type U454 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P454 = PInt<U454>; pub type N454 = NInt<U454>;
pub type U455 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P455 = PInt<U455>; pub type N455 = NInt<U455>;
```

```
pub type U456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P456 = PInt<U456>; pub type N456 = NInt<U456>;
pub type U457 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P457 = PInt<U457>; pub type N457 = NInt<U457>;
pub type U458 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P458 = PInt<U458>; pub type N458 = NInt<U458>;
pub type U459 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P459 = PInt<U459>; pub type N459 = NInt<U459>;
pub type U460 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P460 = PInt<U460>; pub type N460 = NInt<U460>;
pub type U461 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P461 = PInt<U461>; pub type N461 = NInt<U461>;
pub type U462 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P462 = PInt<U462>; pub type N462 = NInt<U462>;
pub type U463 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P463 = PInt<U463>; pub type N463 = NInt<U463>;
pub type U464 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P464 = PInt<U464>; pub type N464 = NInt<U464>;
pub type U465 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P465 = PInt<U465>; pub type N465 = NInt<U465>;
pub type U466 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P466 = PInt<U466>; pub type N466 = NInt<U466>;
pub type U467 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P467 = PInt<U467>; pub type N467 = NInt<U467>;
pub type U468 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P468 = PInt<U468>; pub type N468 = NInt<U468>;
pub type U469 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P469 = PInt<U469>; pub type N469 = NInt<U469>;
pub type U470 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P470 = PInt<U470>; pub type N470 = NInt<U470>;
pub type U471 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P471 = PInt<U471>; pub type N471 = NInt<U471>;
pub type U472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P472 = PInt<U472>; pub type N472 = NInt<U472>;
pub type U473 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P473 = PInt<U473>; pub type N473 = NInt<U473>;
pub type U474 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P474 = PInt<U474>; pub type N474 = NInt<U474>;
pub type U475 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P475 = PInt<U475>; pub type N475 = NInt<U475>;
pub type U476 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P476 = PInt<U476>; pub type N476 = NInt<U476>;
pub type U477 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P477 = PInt<U477>; pub type N477 = NInt<U477>;
pub type U478 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P478 = PInt<U478>; pub type N478 = NInt<U478>;
pub type U479 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P479 = PInt<U479>; pub type N479 = NInt<U479>;
pub type U480 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P480 = PInt<U480>; pub type N480 = NInt<U480>;
pub type U481 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P481 = PInt<U481>; pub type N481 = NInt<U481>;
```

```
pub type U482 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P482 = PInt<U482>; pub type N482 = NInt<U482>;
pub type U483 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P483 = PInt<U483>; pub type N483 = NInt<U483>;
pub type U484 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P484 = PInt<U484>; pub type N484 = NInt<U484>;
pub type U485 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P485 = PInt<U485>; pub type N485 = NInt<U485>;
pub type U486 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P486 = PInt<U486>; pub type N486 = NInt<U486>;
pub type U487 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P487 = PInt<U487>; pub type N487 = NInt<U487>;
pub type U488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P488 = PInt<U488>; pub type N488 = NInt<U488>;
pub type U489 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P489 = PInt<U489>; pub type N489 = NInt<U489>;
pub type U490 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P490 = PInt<U490>; pub type N490 = NInt<U490>;
pub type U491 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P491 = PInt<U491>; pub type N491 = NInt<U491>;
pub type U492 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P492 = PInt<U492>; pub type N492 = NInt<U492>;
pub type U493 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P493 = PInt<U493>; pub type N493 = NInt<U493>;
pub type U494 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P494 = PInt<U494>; pub type N494 = NInt<U494>;
pub type U495 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P495 = PInt<U495>; pub type N495 = NInt<U495>;
pub type U496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P496 = PInt<U496>; pub type N496 = NInt<U496>;
pub type U497 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P497 = PInt<U497>; pub type N497 = NInt<U497>;
pub type U498 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P498 = PInt<U498>; pub type N498 = NInt<U498>;
pub type U499 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P499 = PInt<U499>; pub type N499 = NInt<U499>;
pub type U500 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P500 = PInt<U500>; pub type N500 = NInt<U500>;
pub type U501 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P501 = PInt<U501>; pub type N501 = NInt<U501>;
pub type U502 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P502 = PInt<U502>; pub type N502 = NInt<U502>;
pub type U503 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P503 = PInt<U503>; pub type N503 = NInt<U503>;
pub type U504 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P504 = PInt<U504>; pub type N504 = NInt<U504>;
pub type U505 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P505 = PInt<U505>; pub type N505 = NInt<U505>;
pub type U506 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P506 = PInt<U506>; pub type N506 = NInt<U506>;
pub type U507 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P507 = PInt<U507>; pub type N507 = NInt<U507>;
```

```
pub type U508 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P508 = PInt<U508>; pub type N508 = NInt<U508>;
pub type U509 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P509 = PInt<U509>; pub type N509 = NInt<U509>;
pub type U510 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P510 = PInt<U510>; pub type N510 = NInt<U510>;
pub type U511 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>
pub type P511 = PInt<U511>; pub type N511 = NInt<U511>;
pub type U512 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P512 = PInt<U512>; pub type N512 = NInt<U512>;
pub type U513 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P513 = PInt<U513>; pub type N513 = NInt<U513>;
pub type U514 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P514 = PInt<U514>; pub type N514 = NInt<U514>;
pub type U515 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P515 = PInt<U515>; pub type N515 = NInt<U515>;
pub type U516 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P516 = PInt<U516>; pub type N516 = NInt<U516>;
pub type U517 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P517 = PInt<U517>; pub type N517 = NInt<U517>;
pub type U518 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P518 = PInt<U518>; pub type N518 = NInt<U518>;
pub type U519 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P519 = PInt<U519>; pub type N519 = NInt<U519>;
pub type U520 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P520 = PInt<U520>; pub type N520 = NInt<U520>;
pub type U521 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P521 = PInt<U521>; pub type N521 = NInt<U521>;
pub type U522 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P522 = PInt<U522>; pub type N522 = NInt<U522>;
pub type U523 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P523 = PInt<U523>; pub type N523 = NInt<U523>;
pub type U524 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P524 = PInt<U524>; pub type N524 = NInt<U524>;
pub type U525 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P525 = PInt<U525>; pub type N525 = NInt<U525>;
pub type U526 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P526 = PInt<U526>; pub type N526 = NInt<U526>;
pub type U527 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P527 = PInt<U527>; pub type N527 = NInt<U527>;
pub type U528 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P528 = PInt<U528>; pub type N528 = NInt<U528>;
pub type U529 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P529 = PInt<U529>; pub type N529 = NInt<U529>;
pub type U530 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P530 = PInt<U530>; pub type N530 = NInt<U530>;
pub type U531 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P531 = PInt<U531>; pub type N531 = NInt<U531>;
pub type U532 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P532 = PInt<U532>; pub type N532 = NInt<U532>;
pub type U533 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P533 = PInt<U533>; pub type N533 = NInt<U533>;
```

```
pub type U534 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P534 = PInt<U534>; pub type N534 = NInt<U534>;
pub type U535 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P535 = PInt<U535>; pub type N535 = NInt<U535>;
pub type U536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P536 = PInt<U536>; pub type N536 = NInt<U536>;
pub type U537 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P537 = PInt<U537>; pub type N537 = NInt<U537>;
pub type U538 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P538 = PInt<U538>; pub type N538 = NInt<U538>;
pub type U539 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P539 = PInt<U539>; pub type N539 = NInt<U539>;
pub type U540 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P540 = PInt<U540>; pub type N540 = NInt<U540>;
pub type U541 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P541 = PInt<U541>; pub type N541 = NInt<U541>;
pub type U542 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P542 = PInt<U542>; pub type N542 = NInt<U542>;
pub type U543 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P543 = PInt<U543>; pub type N543 = NInt<U543>;
pub type U544 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P544 = PInt<U544>; pub type N544 = NInt<U544>;
pub type U545 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P545 = PInt<U545>; pub type N545 = NInt<U545>;
pub type U546 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P546 = PInt<U546>; pub type N546 = NInt<U546>;
pub type U547 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P547 = PInt<U547>; pub type N547 = NInt<U547>;
pub type U548 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P548 = PInt<U548>; pub type N548 = NInt<U548>;
pub type U549 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P549 = PInt<U549>; pub type N549 = NInt<U549>;
pub type U550 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P550 = PInt<U550>; pub type N550 = NInt<U550>;
pub type U551 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P551 = PInt<U551>; pub type N551 = NInt<U551>;
pub type U552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P552 = PInt<U552>; pub type N552 = NInt<U552>;
pub type U553 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P553 = PInt<U553>; pub type N553 = NInt<U553>;
pub type U554 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P554 = PInt<U554>; pub type N554 = NInt<U554>;
pub type U555 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P555 = PInt<U555>; pub type N555 = NInt<U555>;
pub type U556 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P556 = PInt<U556>; pub type N556 = NInt<U556>;
pub type U557 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P557 = PInt<U557>; pub type N557 = NInt<U557>;
pub type U558 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P558 = PInt<U558>; pub type N558 = NInt<U558>;
pub type U559 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P559 = PInt<U559>; pub type N559 = NInt<U559>;
```

```
pub type U560 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P560 = PInt<U560>; pub type N560 = NInt<U560>;
pub type U561 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P561 = PInt<U561>; pub type N561 = NInt<U561>;
pub type U562 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P562 = PInt<U562>; pub type N562 = NInt<U562>;
pub type U563 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P563 = PInt<U563>; pub type N563 = NInt<U563>;
pub type U564 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P564 = PInt<U564>; pub type N564 = NInt<U564>;
pub type U565 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P565 = PInt<U565>; pub type N565 = NInt<U565>;
pub type U566 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P566 = PInt<U566>; pub type N566 = NInt<U566>;
pub type U567 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P567 = PInt<U567>; pub type N567 = NInt<U567>;
pub type U568 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P568 = PInt<U568>; pub type N568 = NInt<U568>;
pub type U569 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P569 = PInt<U569>; pub type N569 = NInt<U569>;
pub type U570 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P570 = PInt<U570>; pub type N570 = NInt<U570>;
pub type U571 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P571 = PInt<U571>; pub type N571 = NInt<U571>;
pub type U572 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P572 = PInt<U572>; pub type N572 = NInt<U572>;
pub type U573 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P573 = PInt<U573>; pub type N573 = NInt<U573>;
pub type U574 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P574 = PInt<U574>; pub type N574 = NInt<U574>;
pub type U575 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P575 = PInt<U575>; pub type N575 = NInt<U575>;
pub type U576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P576 = PInt<U576>; pub type N576 = NInt<U576>;
pub type U577 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P577 = PInt<U577>; pub type N577 = NInt<U577>;
pub type U578 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P578 = PInt<U578>; pub type N578 = NInt<U578>;
pub type U579 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P579 = PInt<U579>; pub type N579 = NInt<U579>;
pub type U580 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P580 = PInt<U580>; pub type N580 = NInt<U580>;
pub type U581 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P581 = PInt<U581>; pub type N581 = NInt<U581>;
pub type U582 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P582 = PInt<U582>; pub type N582 = NInt<U582>;
pub type U583 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P583 = PInt<U583>; pub type N583 = NInt<U583>;
pub type U584 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P584 = PInt<U584>; pub type N584 = NInt<U584>;
pub type U585 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P585 = PInt<U585>; pub type N585 = NInt<U585>;
```

```
pub type U586 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P586 = PInt<U586>; pub type N586 = NInt<U586>;
pub type U587 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P587 = PInt<U587>; pub type N587 = NInt<U587>;
pub type U588 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P588 = PInt<U588>; pub type N588 = NInt<U588>;
pub type U589 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P589 = PInt<U589>; pub type N589 = NInt<U589>;
pub type U590 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P590 = PInt<U590>; pub type N590 = NInt<U590>;
pub type U591 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P591 = PInt<U591>; pub type N591 = NInt<U591>;
pub type U592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P592 = PInt<U592>; pub type N592 = NInt<U592>;
pub type U593 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P593 = PInt<U593>; pub type N593 = NInt<U593>;
pub type U594 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P594 = PInt<U594>; pub type N594 = NInt<U594>;
pub type U595 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P595 = PInt<U595>; pub type N595 = NInt<U595>;
pub type U596 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P596 = PInt<U596>; pub type N596 = NInt<U596>;
pub type U597 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P597 = PInt<U597>; pub type N597 = NInt<U597>;
pub type U598 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P598 = PInt<U598>; pub type N598 = NInt<U598>;
pub type U599 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P599 = PInt<U599>; pub type N599 = NInt<U599>;
pub type U600 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P600 = PInt<U600>; pub type N600 = NInt<U600>;
pub type U601 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P601 = PInt<U601>; pub type N601 = NInt<U601>;
pub type U602 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P602 = PInt<U602>; pub type N602 = NInt<U602>;
pub type U603 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P603 = PInt<U603>; pub type N603 = NInt<U603>;
pub type U604 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P604 = PInt<U604>; pub type N604 = NInt<U604>;
pub type U605 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P605 = PInt<U605>; pub type N605 = NInt<U605>;
pub type U606 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P606 = PInt<U606>; pub type N606 = NInt<U606>;
pub type U607 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P607 = PInt<U607>; pub type N607 = NInt<U607>;
pub type U608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P608 = PInt<U608>; pub type N608 = NInt<U608>;
pub type U609 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P609 = PInt<U609>; pub type N609 = NInt<U609>;
pub type U610 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P610 = PInt<U610>; pub type N610 = NInt<U610>;
pub type U611 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P611 = PInt<U611>; pub type N611 = NInt<U611>;
```

```
pub type U612 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P612 = PInt<U612>; pub type N612 = NInt<U612>;
pub type U613 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P613 = PInt<U613>; pub type N613 = NInt<U613>;
pub type U614 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P614 = PInt<U614>; pub type N614 = NInt<U614>;
pub type U615 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P615 = PInt<U615>; pub type N615 = NInt<U615>;
pub type U616 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P616 = PInt<U616>; pub type N616 = NInt<U616>;
pub type U617 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P617 = PInt<U617>; pub type N617 = NInt<U617>;
pub type U618 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P618 = PInt<U618>; pub type N618 = NInt<U618>;
pub type U619 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P619 = PInt<U619>; pub type N619 = NInt<U619>;
pub type U620 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P620 = PInt<U620>; pub type N620 = NInt<U620>;
pub type U621 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P621 = PInt<U621>; pub type N621 = NInt<U621>;
pub type U622 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P622 = PInt<U622>; pub type N622 = NInt<U622>;
pub type U623 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P623 = PInt<U623>; pub type N623 = NInt<U623>;
pub type U624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P624 = PInt<U624>; pub type N624 = NInt<U624>;
pub type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P625 = PInt<U625>; pub type N625 = NInt<U625>;
pub type U626 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P626 = PInt<U626>; pub type N626 = NInt<U626>;
pub type U627 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P627 = PInt<U627>; pub type N627 = NInt<U627>;
pub type U628 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P628 = PInt<U628>; pub type N628 = NInt<U628>;
pub type U629 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P629 = PInt<U629>; pub type N629 = NInt<U629>;
pub type U630 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P630 = PInt<U630>; pub type N630 = NInt<U630>;
pub type U631 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P631 = PInt<U631>; pub type N631 = NInt<U631>;
pub type U632 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P632 = PInt<U632>; pub type N632 = NInt<U632>;
pub type U633 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P633 = PInt<U633>; pub type N633 = NInt<U633>;
pub type U634 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P634 = PInt<U634>; pub type N634 = NInt<U634>;
pub type U635 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P635 = PInt<U635>; pub type N635 = NInt<U635>;
pub type U636 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P636 = PInt<U636>; pub type N636 = NInt<U636>;
pub type U637 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P637 = PInt<U637>; pub type N637 = NInt<U637>;
```

```
pub type U638 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P638 = PInt<U638>; pub type N638 = NInt<U638>;
pub type U639 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P639 = PInt<U639>; pub type N639 = NInt<U639>;
pub type U640 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P640 = PInt<U640>; pub type N640 = NInt<U640>;
pub type U641 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P641 = PInt<U641>; pub type N641 = NInt<U641>;
pub type U642 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P642 = PInt<U642>; pub type N642 = NInt<U642>;
pub type U643 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P643 = PInt<U643>; pub type N643 = NInt<U643>;
pub type U644 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P644 = PInt<U644>; pub type N644 = NInt<U644>;
pub type U645 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P645 = PInt<U645>; pub type N645 = NInt<U645>;
pub type U646 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P646 = PInt<U646>; pub type N646 = NInt<U646>;
pub type U647 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P647 = PInt<U647>; pub type N647 = NInt<U647>;
pub type U648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P648 = PInt<U648>; pub type N648 = NInt<U648>;
pub type U649 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P649 = PInt<U649>; pub type N649 = NInt<U649>;
pub type U650 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P650 = PInt<U650>; pub type N650 = NInt<U650>;
pub type U651 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P651 = PInt<U651>; pub type N651 = NInt<U651>;
pub type U652 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P652 = PInt<U652>; pub type N652 = NInt<U652>;
pub type U653 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P653 = PInt<U653>; pub type N653 = NInt<U653>;
pub type U654 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P654 = PInt<U654>; pub type N654 = NInt<U654>;
pub type U655 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P655 = PInt<U655>; pub type N655 = NInt<U655>;
pub type U656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P656 = PInt<U656>; pub type N656 = NInt<U656>;
pub type U657 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P657 = PInt<U657>; pub type N657 = NInt<U657>;
pub type U658 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P658 = PInt<U658>; pub type N658 = NInt<U658>;
pub type U659 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P659 = PInt<U659>; pub type N659 = NInt<U659>;
pub type U660 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P660 = PInt<U660>; pub type N660 = NInt<U660>;
pub type U661 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P661 = PInt<U661>; pub type N661 = NInt<U661>;
pub type U662 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P662 = PInt<U662>; pub type N662 = NInt<U662>;
pub type U663 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P663 = PInt<U663>; pub type N663 = NInt<U663>;
```

```
pub type U664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P664 = PInt<U664>; pub type N664 = NInt<U664>;
pub type U665 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P665 = PInt<U665>; pub type N665 = NInt<U665>;
pub type U666 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P666 = PInt<U666>; pub type N666 = NInt<U666>;
pub type U667 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P667 = PInt<U667>; pub type N667 = NInt<U667>;
pub type U668 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P668 = PInt<U668>; pub type N668 = NInt<U668>;
pub type U669 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P669 = PInt<U669>; pub type N669 = NInt<U669>;
pub type U670 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P670 = PInt<U670>; pub type N670 = NInt<U670>;
pub type U671 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P671 = PInt<U671>; pub type N671 = NInt<U671>;
pub type U672 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P672 = PInt<U672>; pub type N672 = NInt<U672>;
pub type U673 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P673 = PInt<U673>; pub type N673 = NInt<U673>;
pub type U674 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P674 = PInt<U674>; pub type N674 = NInt<U674>;
pub type U675 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P675 = PInt<U675>; pub type N675 = NInt<U675>;
pub type U676 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P676 = PInt<U676>; pub type N676 = NInt<U676>;
pub type U677 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P677 = PInt<U677>; pub type N677 = NInt<U677>;
pub type U678 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P678 = PInt<U678>; pub type N678 = NInt<U678>;
pub type U679 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P679 = PInt<U679>; pub type N679 = NInt<U679>;
pub type U680 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P680 = PInt<U680>; pub type N680 = NInt<U680>;
pub type U681 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P681 = PInt<U681>; pub type N681 = NInt<U681>;
pub type U682 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P682 = PInt<U682>; pub type N682 = NInt<U682>;
pub type U683 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P683 = PInt<U683>; pub type N683 = NInt<U683>;
pub type U684 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P684 = PInt<U684>; pub type N684 = NInt<U684>;
pub type U685 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P685 = PInt<U685>; pub type N685 = NInt<U685>;
pub type U686 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P686 = PInt<U686>; pub type N686 = NInt<U686>;
pub type U687 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P687 = PInt<U687>; pub type N687 = NInt<U687>;
pub type U688 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P688 = PInt<U688>; pub type N688 = NInt<U688>;
pub type U689 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P689 = PInt<U689>; pub type N689 = NInt<U689>;
```

```
pub type U690 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P690 = PInt<U690>; pub type N690 = NInt<U690>;
pub type U691 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P691 = PInt<U691>; pub type N691 = NInt<U691>;
pub type U692 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P692 = PInt<U692>; pub type N692 = NInt<U692>;
pub type U693 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P693 = PInt<U693>; pub type N693 = NInt<U693>;
pub type U694 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P694 = PInt<U694>; pub type N694 = NInt<U694>;
pub type U695 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P695 = PInt<U695>; pub type N695 = NInt<U695>;
pub type U696 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P696 = PInt<U696>; pub type N696 = NInt<U696>;
pub type U697 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P697 = PInt<U697>; pub type N697 = NInt<U697>;
pub type U698 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P698 = PInt<U698>; pub type N698 = NInt<U698>;
pub type U699 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P699 = PInt<U699>; pub type N699 = NInt<U699>;
pub type U700 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P700 = PInt<U700>; pub type N700 = NInt<U700>;
pub type U701 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P701 = PInt<U701>; pub type N701 = NInt<U701>;
pub type U702 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P702 = PInt<U702>; pub type N702 = NInt<U702>;
pub type U703 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P703 = PInt<U703>; pub type N703 = NInt<U703>;
pub type U704 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P704 = PInt<U704>; pub type N704 = NInt<U704>;
pub type U705 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P705 = PInt<U705>; pub type N705 = NInt<U705>;
pub type U706 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P706 = PInt<U706>; pub type N706 = NInt<U706>;
pub type U707 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P707 = PInt<U707>; pub type N707 = NInt<U707>;
pub type U708 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P708 = PInt<U708>; pub type N708 = NInt<U708>;
pub type U709 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P709 = PInt<U709>; pub type N709 = NInt<U709>;
pub type U710 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P710 = PInt<U710>; pub type N710 = NInt<U710>;
pub type U711 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P711 = PInt<U711>; pub type N711 = NInt<U711>;
pub type U712 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P712 = PInt<U712>; pub type N712 = NInt<U712>;
pub type U713 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P713 = PInt<U713>; pub type N713 = NInt<U713>;
pub type U714 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P714 = PInt<U714>; pub type N714 = NInt<U714>;
pub type U715 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P715 = PInt<U715>; pub type N715 = NInt<U715>;
```

```
pub type U716 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P716 = PInt<U716>; pub type N716 = NInt<U716>;
pub type U717 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P717 = PInt<U717>; pub type N717 = NInt<U717>;
pub type U718 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P718 = PInt<U718>; pub type N718 = NInt<U718>;
pub type U719 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P719 = PInt<U719>; pub type N719 = NInt<U719>;
pub type U720 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P720 = PInt<U720>; pub type N720 = NInt<U720>;
pub type U721 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P721 = PInt<U721>; pub type N721 = NInt<U721>;
pub type U722 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P722 = PInt<U722>; pub type N722 = NInt<U722>;
pub type U723 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P723 = PInt<U723>; pub type N723 = NInt<U723>;
pub type U724 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P724 = PInt<U724>; pub type N724 = NInt<U724>;
pub type U725 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P725 = PInt<U725>; pub type N725 = NInt<U725>;
pub type U726 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P726 = PInt<U726>; pub type N726 = NInt<U726>;
pub type U727 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P727 = PInt<U727>; pub type N727 = NInt<U727>;
pub type U728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P728 = PInt<U728>; pub type N728 = NInt<U728>;
pub type U729 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P729 = PInt<U729>; pub type N729 = NInt<U729>;
pub type U730 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P730 = PInt<U730>; pub type N730 = NInt<U730>;
pub type U731 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P731 = PInt<U731>; pub type N731 = NInt<U731>;
pub type U732 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P732 = PInt<U732>; pub type N732 = NInt<U732>;
pub type U733 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P733 = PInt<U733>; pub type N733 = NInt<U733>;
pub type U734 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P734 = PInt<U734>; pub type N734 = NInt<U734>;
pub type U735 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P735 = PInt<U735>; pub type N735 = NInt<U735>;
pub type U736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P736 = PInt<U736>; pub type N736 = NInt<U736>;
pub type U737 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P737 = PInt<U737>; pub type N737 = NInt<U737>;
pub type U738 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P738 = PInt<U738>; pub type N738 = NInt<U738>;
pub type U739 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P739 = PInt<U739>; pub type N739 = NInt<U739>;
pub type U740 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P740 = PInt<U740>; pub type N740 = NInt<U740>;
pub type U741 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P741 = PInt<U741>; pub type N741 = NInt<U741>;
```

```
pub type U742 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P742 = PInt<U742>; pub type N742 = NInt<U742>;
pub type U743 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P743 = PInt<U743>; pub type N743 = NInt<U743>;
pub type U744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P744 = PInt<U744>; pub type N744 = NInt<U744>;
pub type U745 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P745 = PInt<U745>; pub type N745 = NInt<U745>;
pub type U746 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P746 = PInt<U746>; pub type N746 = NInt<U746>;
pub type U747 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P747 = PInt<U747>; pub type N747 = NInt<U747>;
pub type U748 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P748 = PInt<U748>; pub type N748 = NInt<U748>;
pub type U749 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P749 = PInt<U749>; pub type N749 = NInt<U749>;
pub type U750 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P750 = PInt<U750>; pub type N750 = NInt<U750>;
pub type U751 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P751 = PInt<U751>; pub type N751 = NInt<U751>;
pub type U752 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P752 = PInt<U752>; pub type N752 = NInt<U752>;
pub type U753 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P753 = PInt<U753>; pub type N753 = NInt<U753>;
pub type U754 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P754 = PInt<U754>; pub type N754 = NInt<U754>;
pub type U755 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P755 = PInt<U755>; pub type N755 = NInt<U755>;
pub type U756 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P756 = PInt<U756>; pub type N756 = NInt<U756>;
pub type U757 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P757 = PInt<U757>; pub type N757 = NInt<U757>;
pub type U758 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P758 = PInt<U758>; pub type N758 = NInt<U758>;
pub type U759 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P759 = PInt<U759>; pub type N759 = NInt<U759>;
pub type U760 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P760 = PInt<U760>; pub type N760 = NInt<U760>;
pub type U761 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P761 = PInt<U761>; pub type N761 = NInt<U761>;
pub type U762 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P762 = PInt<U762>; pub type N762 = NInt<U762>;
pub type U763 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P763 = PInt<U763>; pub type N763 = NInt<U763>;
pub type U764 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P764 = PInt<U764>; pub type N764 = NInt<U764>;
pub type U765 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P765 = PInt<U765>; pub type N765 = NInt<U765>;
pub type U766 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P766 = PInt<U766>; pub type N766 = NInt<U766>;
pub type U767 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P767 = PInt<U767>; pub type N767 = NInt<U767>;
```

```
pub type U768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P768 = PInt<U768>; pub type N768 = NInt<U768>;
pub type U769 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P769 = PInt<U769>; pub type N769 = NInt<U769>;
pub type U770 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P770 = PInt<U770>; pub type N770 = NInt<U770>;
pub type U771 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P771 = PInt<U771>; pub type N771 = NInt<U771>;
pub type U772 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P772 = PInt<U772>; pub type N772 = NInt<U772>;
pub type U773 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P773 = PInt<U773>; pub type N773 = NInt<U773>;
pub type U774 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P774 = PInt<U774>; pub type N774 = NInt<U774>;
pub type U775 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P775 = PInt<U775>; pub type N775 = NInt<U775>;
pub type U776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P776 = PInt<U776>; pub type N776 = NInt<U776>;
pub type U777 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P777 = PInt<U777>; pub type N777 = NInt<U777>;
pub type U778 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P778 = PInt<U778>; pub type N778 = NInt<U778>;
pub type U779 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P779 = PInt<U779>; pub type N779 = NInt<U779>;
pub type U780 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P780 = PInt<U780>; pub type N780 = NInt<U780>;
pub type U781 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P781 = PInt<U781>; pub type N781 = NInt<U781>;
pub type U782 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P782 = PInt<U782>; pub type N782 = NInt<U782>;
pub type U783 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P783 = PInt<U783>; pub type N783 = NInt<U783>;
pub type U784 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P784 = PInt<U784>; pub type N784 = NInt<U784>;
pub type U785 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P785 = PInt<U785>; pub type N785 = NInt<U785>;
pub type U786 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P786 = PInt<U786>; pub type N786 = NInt<U786>;
pub type U787 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P787 = PInt<U787>; pub type N787 = NInt<U787>;
pub type U788 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P788 = PInt<U788>; pub type N788 = NInt<U788>;
pub type U789 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P789 = PInt<U789>; pub type N789 = NInt<U789>;
pub type U790 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P790 = PInt<U790>; pub type N790 = NInt<U790>;
pub type U791 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P791 = PInt<U791>; pub type N791 = NInt<U791>;
pub type U792 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P792 = PInt<U792>; pub type N792 = NInt<U792>;
pub type U793 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P793 = PInt<U793>; pub type N793 = NInt<U793>;
```

```
pub type U794 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P794 = PInt<U794>; pub type N794 = NInt<U794>;
pub type U795 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P795 = PInt<U795>; pub type N795 = NInt<U795>;
pub type U796 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P796 = PInt<U796>; pub type N796 = NInt<U796>;
pub type U797 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P797 = PInt<U797>; pub type N797 = NInt<U797>;
pub type U798 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P798 = PInt<U798>; pub type N798 = NInt<U798>;
pub type U799 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P799 = PInt<U799>; pub type N799 = NInt<U799>;
pub type U800 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P800 = PInt<U800>; pub type N800 = NInt<U800>;
pub type U801 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P801 = PInt<U801>; pub type N801 = NInt<U801>;
pub type U802 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P802 = PInt<U802>; pub type N802 = NInt<U802>;
pub type U803 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P803 = PInt<U803>; pub type N803 = NInt<U803>;
pub type U804 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P804 = PInt<U804>; pub type N804 = NInt<U804>;
pub type U805 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P805 = PInt<U805>; pub type N805 = NInt<U805>;
pub type U806 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P806 = PInt<U806>; pub type N806 = NInt<U806>;
pub type U807 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P807 = PInt<U807>; pub type N807 = NInt<U807>;
pub type U808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P808 = PInt<U808>; pub type N808 = NInt<U808>;
pub type U809 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P809 = PInt<U809>; pub type N809 = NInt<U809>;
pub type U810 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P810 = PInt<U810>; pub type N810 = NInt<U810>;
pub type U811 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P811 = PInt<U811>; pub type N811 = NInt<U811>;
pub type U812 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P812 = PInt<U812>; pub type N812 = NInt<U812>;
pub type U813 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P813 = PInt<U813>; pub type N813 = NInt<U813>;
pub type U814 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P814 = PInt<U814>; pub type N814 = NInt<U814>;
pub type U815 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P815 = PInt<U815>; pub type N815 = NInt<U815>;
pub type U816 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P816 = PInt<U816>; pub type N816 = NInt<U816>;
pub type U817 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P817 = PInt<U817>; pub type N817 = NInt<U817>;
pub type U818 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P818 = PInt<U818>; pub type N818 = NInt<U818>;
pub type U819 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P819 = PInt<U819>; pub type N819 = NInt<U819>;
```

```
pub type U820 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P820 = PInt<U820>; pub type N820 = NInt<U820>;
pub type U821 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P821 = PInt<U821>; pub type N821 = NInt<U821>;
pub type U822 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P822 = PInt<U822>; pub type N822 = NInt<U822>;
pub type U823 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P823 = PInt<U823>; pub type N823 = NInt<U823>;
pub type U824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P824 = PInt<U824>; pub type N824 = NInt<U824>;
pub type U825 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P825 = PInt<U825>; pub type N825 = NInt<U825>;
pub type U826 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P826 = PInt<U826>; pub type N826 = NInt<U826>;
pub type U827 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P827 = PInt<U827>; pub type N827 = NInt<U827>;
pub type U828 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P828 = PInt<U828>; pub type N828 = NInt<U828>;
pub type U829 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P829 = PInt<U829>; pub type N829 = NInt<U829>;
pub type U830 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P830 = PInt<U830>; pub type N830 = NInt<U830>;
pub type U831 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P831 = PInt<U831>; pub type N831 = NInt<U831>;
pub type U832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P832 = PInt<U832>; pub type N832 = NInt<U832>;
pub type U833 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P833 = PInt<U833>; pub type N833 = NInt<U833>;
pub type U834 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P834 = PInt<U834>; pub type N834 = NInt<U834>;
pub type U835 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P835 = PInt<U835>; pub type N835 = NInt<U835>;
pub type U836 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P836 = PInt<U836>; pub type N836 = NInt<U836>;
pub type U837 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P837 = PInt<U837>; pub type N837 = NInt<U837>;
pub type U838 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P838 = PInt<U838>; pub type N838 = NInt<U838>;
pub type U839 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P839 = PInt<U839>; pub type N839 = NInt<U839>;
pub type U840 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P840 = PInt<U840>; pub type N840 = NInt<U840>;
pub type U841 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P841 = PInt<U841>; pub type N841 = NInt<U841>;
pub type U842 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P842 = PInt<U842>; pub type N842 = NInt<U842>;
pub type U843 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P843 = PInt<U843>; pub type N843 = NInt<U843>;
pub type U844 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P844 = PInt<U844>; pub type N844 = NInt<U844>;
pub type U845 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P845 = PInt<U845>; pub type N845 = NInt<U845>;
```

```
pub type U846 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P846 = PInt<U846>; pub type N846 = NInt<U846>;
pub type U847 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P847 = PInt<U847>; pub type N847 = NInt<U847>;
pub type U848 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P848 = PInt<U848>; pub type N848 = NInt<U848>;
pub type U849 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P849 = PInt<U849>; pub type N849 = NInt<U849>;
pub type U850 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P850 = PInt<U850>; pub type N850 = NInt<U850>;
pub type U851 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P851 = PInt<U851>; pub type N851 = NInt<U851>;
pub type U852 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P852 = PInt<U852>; pub type N852 = NInt<U852>;
pub type U853 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P853 = PInt<U853>; pub type N853 = NInt<U853>;
pub type U854 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P854 = PInt<U854>; pub type N854 = NInt<U854>;
pub type U855 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P855 = PInt<U855>; pub type N855 = NInt<U855>;
pub type U856 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P856 = PInt<U856>; pub type N856 = NInt<U856>;
pub type U857 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P857 = PInt<U857>; pub type N857 = NInt<U857>;
pub type U858 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P858 = PInt<U858>; pub type N858 = NInt<U858>;
pub type U859 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P859 = PInt<U859>; pub type N859 = NInt<U859>;
pub type U860 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P860 = PInt<U860>; pub type N860 = NInt<U860>;
pub type U861 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P861 = PInt<U861>; pub type N861 = NInt<U861>;
pub type U862 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P862 = PInt<U862>; pub type N862 = NInt<U862>;
pub type U863 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P863 = PInt<U863>; pub type N863 = NInt<U863>;
pub type U864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P864 = PInt<U864>; pub type N864 = NInt<U864>;
pub type U865 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P865 = PInt<U865>; pub type N865 = NInt<U865>;
pub type U866 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P866 = PInt<U866>; pub type N866 = NInt<U866>;
pub type U867 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P867 = PInt<U867>; pub type N867 = NInt<U867>;
pub type U868 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P868 = PInt<U868>; pub type N868 = NInt<U868>;
pub type U869 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P869 = PInt<U869>; pub type N869 = NInt<U869>;
pub type U870 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P870 = PInt<U870>; pub type N870 = NInt<U870>;
pub type U871 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P871 = PInt<U871>; pub type N871 = NInt<U871>;
```

```
pub type U872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P872 = PInt<U872>; pub type N872 = NInt<U872>;
pub type U873 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P873 = PInt<U873>; pub type N873 = NInt<U873>;
pub type U874 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P874 = PInt<U874>; pub type N874 = NInt<U874>;
pub type U875 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P875 = PInt<U875>; pub type N875 = NInt<U875>;
pub type U876 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P876 = PInt<U876>; pub type N876 = NInt<U876>;
pub type U877 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P877 = PInt<U877>; pub type N877 = NInt<U877>;
pub type U878 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P878 = PInt<U878>; pub type N878 = NInt<U878>;
pub type U879 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P879 = PInt<U879>; pub type N879 = NInt<U879>;
pub type U880 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P880 = PInt<U880>; pub type N880 = NInt<U880>;
pub type U881 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P881 = PInt<U881>; pub type N881 = NInt<U881>;
pub type U882 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P882 = PInt<U882>; pub type N882 = NInt<U882>;
pub type U883 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P883 = PInt<U883>; pub type N883 = NInt<U883>;
pub type U884 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P884 = PInt<U884>; pub type N884 = NInt<U884>;
pub type U885 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P885 = PInt<U885>; pub type N885 = NInt<U885>;
pub type U886 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P886 = PInt<U886>; pub type N886 = NInt<U886>;
pub type U887 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P887 = PInt<U887>; pub type N887 = NInt<U887>;
pub type U888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P888 = PInt<U888>; pub type N888 = NInt<U888>;
pub type U889 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P889 = PInt<U889>; pub type N889 = NInt<U889>;
pub type U890 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P890 = PInt<U890>; pub type N890 = NInt<U890>;
pub type U891 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P891 = PInt<U891>; pub type N891 = NInt<U891>;
pub type U892 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P892 = PInt<U892>; pub type N892 = NInt<U892>;
pub type U893 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P893 = PInt<U893>; pub type N893 = NInt<U893>;
pub type U894 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P894 = PInt<U894>; pub type N894 = NInt<U894>;
pub type U895 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P895 = PInt<U895>; pub type N895 = NInt<U895>;
pub type U896 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P896 = PInt<U896>; pub type N896 = NInt<U896>;
pub type U897 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P897 = PInt<U897>; pub type N897 = NInt<U897>;
```

```
pub type U898 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P898 = PInt<U898>; pub type N898 = NInt<U898>;
pub type U899 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P899 = PInt<U899>; pub type N899 = NInt<U899>;
pub type U900 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P900 = PInt<U900>; pub type N900 = NInt<U900>;
pub type U901 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P901 = PInt<U901>; pub type N901 = NInt<U901>;
pub type U902 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P902 = PInt<U902>; pub type N902 = NInt<U902>;
pub type U903 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P903 = PInt<U903>; pub type N903 = NInt<U903>;
pub type U904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P904 = PInt<U904>; pub type N904 = NInt<U904>;
pub type U905 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P905 = PInt<U905>; pub type N905 = NInt<U905>;
pub type U906 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P906 = PInt<U906>; pub type N906 = NInt<U906>;
pub type U907 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P907 = PInt<U907>; pub type N907 = NInt<U907>;
pub type U908 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P908 = PInt<U908>; pub type N908 = NInt<U908>;
pub type U909 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P909 = PInt<U909>; pub type N909 = NInt<U909>;
pub type U910 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P910 = PInt<U910>; pub type N910 = NInt<U910>;
pub type U911 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P911 = PInt<U911>; pub type N911 = NInt<U911>;
pub type U912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P912 = PInt<U912>; pub type N912 = NInt<U912>;
pub type U913 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P913 = PInt<U913>; pub type N913 = NInt<U913>;
pub type U914 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P914 = PInt<U914>; pub type N914 = NInt<U914>;
pub type U915 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P915 = PInt<U915>; pub type N915 = NInt<U915>;
pub type U916 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P916 = PInt<U916>; pub type N916 = NInt<U916>;
pub type U917 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P917 = PInt<U917>; pub type N917 = NInt<U917>;
pub type U918 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P918 = PInt<U918>; pub type N918 = NInt<U918>;
pub type U919 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P919 = PInt<U919>; pub type N919 = NInt<U919>;
pub type U920 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P920 = PInt<U920>; pub type N920 = NInt<U920>;
pub type U921 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P921 = PInt<U921>; pub type N921 = NInt<U921>;
pub type U922 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P922 = PInt<U922>; pub type N922 = NInt<U922>;
pub type U923 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P923 = PInt<U923>; pub type N923 = NInt<U923>;
```

```
pub type U924 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P924 = PInt<U924>; pub type N924 = NInt<U924>;
pub type U925 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P925 = PInt<U925>; pub type N925 = NInt<U925>;
pub type U926 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P926 = PInt<U926>; pub type N926 = NInt<U926>;
pub type U927 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P927 = PInt<U927>; pub type N927 = NInt<U927>;
pub type U928 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P928 = PInt<U928>; pub type N928 = NInt<U928>;
pub type U929 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P929 = PInt<U929>; pub type N929 = NInt<U929>;
pub type U930 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P930 = PInt<U930>; pub type N930 = NInt<U930>;
pub type U931 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P931 = PInt<U931>; pub type N931 = NInt<U931>;
pub type U932 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P932 = PInt<U932>; pub type N932 = NInt<U932>;
pub type U933 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P933 = PInt<U933>; pub type N933 = NInt<U933>;
pub type U934 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P934 = PInt<U934>; pub type N934 = NInt<U934>;
pub type U935 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P935 = PInt<U935>; pub type N935 = NInt<U935>;
pub type U936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P936 = PInt<U936>; pub type N936 = NInt<U936>;
pub type U937 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P937 = PInt<U937>; pub type N937 = NInt<U937>;
pub type U938 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P938 = PInt<U938>; pub type N938 = NInt<U938>;
pub type U939 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P939 = PInt<U939>; pub type N939 = NInt<U939>;
pub type U940 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P940 = PInt<U940>; pub type N940 = NInt<U940>;
pub type U941 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P941 = PInt<U941>; pub type N941 = NInt<U941>;
pub type U942 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P942 = PInt<U942>; pub type N942 = NInt<U942>;
pub type U943 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P943 = PInt<U943>; pub type N943 = NInt<U943>;
pub type U944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P944 = PInt<U944>; pub type N944 = NInt<U944>;
pub type U945 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P945 = PInt<U945>; pub type N945 = NInt<U945>;
pub type U946 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P946 = PInt<U946>; pub type N946 = NInt<U946>;
pub type U947 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P947 = PInt<U947>; pub type N947 = NInt<U947>;
pub type U948 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P948 = PInt<U948>; pub type N948 = NInt<U948>;
pub type U949 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P949 = PInt<U949>; pub type N949 = NInt<U949>;
```

```
pub type U950 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P950 = PInt<U950>; pub type N950 = NInt<U950>;
pub type U951 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P951 = PInt<U951>; pub type N951 = NInt<U951>;
pub type U952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P952 = PInt<U952>; pub type N952 = NInt<U952>;
pub type U953 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P953 = PInt<U953>; pub type N953 = NInt<U953>;
pub type U954 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P954 = PInt<U954>; pub type N954 = NInt<U954>;
pub type U955 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P955 = PInt<U955>; pub type N955 = NInt<U955>;
pub type U956 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P956 = PInt<U956>; pub type N956 = NInt<U956>;
pub type U957 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P957 = PInt<U957>; pub type N957 = NInt<U957>;
pub type U958 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P958 = PInt<U958>; pub type N958 = NInt<U958>;
pub type U959 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P959 = PInt<U959>; pub type N959 = NInt<U959>;
pub type U960 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P960 = PInt<U960>; pub type N960 = NInt<U960>;
pub type U961 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P961 = PInt<U961>; pub type N961 = NInt<U961>;
pub type U962 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P962 = PInt<U962>; pub type N962 = NInt<U962>;
pub type U963 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P963 = PInt<U963>; pub type N963 = NInt<U963>;
pub type U964 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P964 = PInt<U964>; pub type N964 = NInt<U964>;
pub type U965 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P965 = PInt<U965>; pub type N965 = NInt<U965>;
pub type U966 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P966 = PInt<U966>; pub type N966 = NInt<U966>;
pub type U967 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P967 = PInt<U967>; pub type N967 = NInt<U967>;
pub type U968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P968 = PInt<U968>; pub type N968 = NInt<U968>;
pub type U969 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P969 = PInt<U969>; pub type N969 = NInt<U969>;
pub type U970 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P970 = PInt<U970>; pub type N970 = NInt<U970>;
pub type U971 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P971 = PInt<U971>; pub type N971 = NInt<U971>;
pub type U972 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P972 = PInt<U972>; pub type N972 = NInt<U972>;
pub type U973 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P973 = PInt<U973>; pub type N973 = NInt<U973>;
pub type U974 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P974 = PInt<U974>; pub type N974 = NInt<U974>;
pub type U975 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P975 = PInt<U975>; pub type N975 = NInt<U975>;
```

```
pub type U976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P976 = PInt<U976>; pub type N976 = NInt<U976>;
pub type U977 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P977 = PInt<U977>; pub type N977 = NInt<U977>;
pub type U978 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P978 = PInt<U978>; pub type N978 = NInt<U978>;
pub type U979 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P979 = PInt<U979>; pub type N979 = NInt<U979>;
pub type U980 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P980 = PInt<U980>; pub type N980 = NInt<U980>;
pub type U981 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P981 = PInt<U981>; pub type N981 = NInt<U981>;
pub type U982 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P982 = PInt<U982>; pub type N982 = NInt<U982>;
pub type U983 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P983 = PInt<U983>; pub type N983 = NInt<U983>;
pub type U984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P984 = PInt<U984>; pub type N984 = NInt<U984>;
pub type U985 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P985 = PInt<U985>; pub type N985 = NInt<U985>;
pub type U986 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P986 = PInt<U986>; pub type N986 = NInt<U986>;
pub type U987 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P987 = PInt<U987>; pub type N987 = NInt<U987>;
pub type U988 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P988 = PInt<U988>; pub type N988 = NInt<U988>;
pub type U989 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P989 = PInt<U989>; pub type N989 = NInt<U989>;
pub type U990 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P990 = PInt<U990>; pub type N990 = NInt<U990>;
pub type U991 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P991 = PInt<U991>; pub type N991 = NInt<U991>;
pub type U992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P992 = PInt<U992>; pub type N992 = NInt<U992>;
pub type U993 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P993 = PInt<U993>; pub type N993 = NInt<U993>;
pub type U994 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P994 = PInt<U994>; pub type N994 = NInt<U994>;
pub type U995 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P995 = PInt<U995>; pub type N995 = NInt<U995>;
pub type U996 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P996 = PInt<U996>; pub type N996 = NInt<U996>;
pub type U997 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P997 = PInt<U997>; pub type N997 = NInt<U997>;
pub type U998 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P998 = PInt<U998>; pub type N998 = NInt<U998>;
pub type U999 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm
pub type P999 = PInt<U999>; pub type N999 = NInt<U999>;
pub type U1000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1000 = PInt<U1000>; pub type N1000 = NInt<U1000>;
pub type U1001 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1001 = PInt<U1001>; pub type N1001 = NInt<U1001>;
```

```
pub type U1002 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1002 = PInt<U1002>; pub type N1002 = NInt<U1002>;
pub type U1003 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1003 = PInt<U1003>; pub type N1003 = NInt<U1003>;
pub type U1004 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1004 = PInt<U1004>; pub type N1004 = NInt<U1004>;
pub type U1005 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1005 = PInt<U1005>; pub type N1005 = NInt<U1005>;
pub type U1006 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1006 = PInt<U1006>; pub type N1006 = NInt<U1006>;
pub type U1007 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1007 = PInt<U1007>; pub type N1007 = NInt<U1007>;
pub type U1008 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1008 = PInt<U1008>; pub type N1008 = NInt<U1008>;
pub type U1009 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1009 = PInt<U1009>; pub type N1009 = NInt<U1009>;
pub type U1010 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1010 = PInt<U1010>; pub type N1010 = NInt<U1010>;
pub type U1011 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1011 = PInt<U1011>; pub type N1011 = NInt<U1011>;
pub type U1012 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1012 = PInt<U1012>; pub type N1012 = NInt<U1012>;
pub type U1013 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1013 = PInt<U1013>; pub type N1013 = NInt<U1013>;
pub type U1014 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1014 = PInt<U1014>; pub type N1014 = NInt<U1014>;
pub type U1015 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1015 = PInt<U1015>; pub type N1015 = NInt<U1015>;
pub type U1016 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1016 = PInt<U1016>; pub type N1016 = NInt<U1016>;
pub type U1017 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1017 = PInt<U1017>; pub type N1017 = NInt<U1017>;
pub type U1018 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1018 = PInt<U1018>; pub type N1018 = NInt<U1018>;
pub type U1019 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1019 = PInt<U1019>; pub type N1019 = NInt<U1019>;
pub type U1020 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1020 = PInt<U1020>; pub type N1020 = NInt<U1020>;
pub type U1021 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1021 = PInt<U1021>; pub type N1021 = NInt<U1021>;
pub type U1022 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1022 = PInt<U1022>; pub type N1022 = NInt<U1022>;
pub type U1023 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer
pub type P1023 = PInt<U1023>; pub type N1023 = NInt<U1023>;
pub type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1024 = PInt<U1024>; pub type N1024 = NInt<U1024>;
pub type U2048 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2048 = PInt<U2048>; pub type N2048 = NInt<U2048>;
pub type U4096 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4096 = PInt<U4096>; pub type N4096 = NInt<U4096>;
pub type U8192 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P8192 = PInt<U8192>; pub type N8192 = NInt<U8192>;
```

```
pub type U16384 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P16384 = PInt<U16384>; pub type N16384 = NInt<U16384>;
pub type U32768 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P32768 = PInt<U32768>; pub type N32768 = NInt<U32768>;
pub type U65536 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P65536 = PInt<U65536>; pub type N65536 = NInt<U65536>;
pub type U131072 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P131072 = PInt<U131072>; pub type N131072 = NInt<U131072>;
pub type U262144 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P262144 = PInt<U262144>; pub type N262144 = NInt<U262144>;
pub type U524288 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P524288 = PInt<U524288>; pub type N524288 = NInt<U524288>;
pub type U1048576 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P1048576 = PInt<U1048576>; pub type N1048576 = NInt<U1048576>;
pub type U2097152 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P2097152 = PInt<U2097152>; pub type N2097152 = NInt<U2097152>;
pub type U4194304 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P4194304 = PInt<U4194304>; pub type N4194304 = NInt<U4194304>;
pub type U8388608 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P8388608 = PInt<U8388608>; pub type N8388608 = NInt<U8388608>;
pub type U16777216 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P16777216 = PInt<U16777216>; pub type N16777216 = NInt<U167772
pub type U33554432 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P33554432 = PInt<U33554432>; pub type N33554432 = NInt<U335544
pub type U67108864 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P67108864 = PInt<U67108864>; pub type N67108864 = NInt<U671088
pub type U134217728 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P134217728 = PInt<U134217728>; pub type N134217728 = NInt<U134
pub type U268435456 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P268435456 = PInt<U268435456>; pub type N268435456 = NInt<U268
pub type U536870912 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P536870912 = PInt<U536870912>; pub type N536870912 = NInt<U536
pub type U1073741824 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P1073741824 = PInt<U1073741824>; pub type N1073741824 = NInt<U
pub type U2147483648 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P2147483648 = PInt<U2147483648>; pub type N2147483648 = NInt<U
pub type U4294967296 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P4294967296 = PInt<U4294967296>; pub type N4294967296 = NInt<U
pub type U8589934592 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P8589934592 = PInt<U8589934592>; pub type N8589934592 = NInt<U
pub type U17179869184 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P17179869184 = PInt<U17179869184>; pub type N17179869184 = NIn
pub type U34359738368 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P34359738368 = PInt<U34359738368>; pub type N34359738368 = NIn
pub type U68719476736 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P68719476736 = PInt<U68719476736>; pub type N68719476736 = NIn
pub type U137438953472 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P137438953472 = PInt<U137438953472>; pub type N137438953472 =
pub type U274877906944 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P274877906944 = PInt<U274877906944>; pub type N274877906944 =
pub type U549755813888 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P549755813888 = PInt<U549755813888>; pub type N549755813888 =
```

```
pub type U1099511627776 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P1099511627776 = PInt<U1099511627776>; pub type N1099511627776
pub type U2199023255552 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P2199023255552 = PInt<U2199023255552>; pub type N2199023255552
pub type U4398046511104 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P4398046511104 = PInt<U4398046511104>; pub type N4398046511104
pub type U8796093022208 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P8796093022208 = PInt<U8796093022208>; pub type N8796093022208
pub type U17592186044416 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P17592186044416 = PInt<U17592186044416>; pub type N17592186044
pub type U35184372088832 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P35184372088832 = PInt<U35184372088832>; pub type N35184372088
pub type U70368744177664 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P70368744177664 = PInt<U70368744177664>; pub type N70368744177
pub type U140737488355328 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P140737488355328 = PInt<U140737488355328>; pub type N140737488
pub type U281474976710656 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P281474976710656 = PInt<U281474976710656>; pub type N281474976
pub type U562949953421312 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P562949953421312 = PInt<U562949953421312>; pub type N562949953
pub type U1125899906842624 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P1125899906842624 = PInt<U1125899906842624>; pub type N1125899
pub type U2251799813685248 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P2251799813685248 = PInt<U2251799813685248>; pub type N2251799
pub type U4503599627370496 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P4503599627370496 = PInt<U4503599627370496>; pub type N4503599
pub type U9007199254740992 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P9007199254740992 = PInt<U9007199254740992>; pub type N9007199
pub type U18014398509481984 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P18014398509481984 = PInt<U18014398509481984>; pub type N18014
pub type U36028797018963968 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P36028797018963968 = PInt<U36028797018963968>; pub type N36028
pub type U72057594037927936 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
pub type P72057594037927936 = PInt<U72057594037927936>; pub type N72057
pub type U144115188075855872 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P144115188075855872 = PInt<U144115188075855872>; pub type N144
pub type U288230376151711744 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P288230376151711744 = PInt<U288230376151711744>; pub type N288
pub type U576460752303423488 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
pub type P576460752303423488 = PInt<U576460752303423488>; pub type N576
pub type U1152921504606846976 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P1152921504606846976 = PInt<U1152921504606846976>; pub type N1
pub type U2305843009213693952 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P2305843009213693952 = PInt<U2305843009213693952>; pub type N2
pub type U4611686018427387904 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type P4611686018427387904 = PInt<U4611686018427387904>; pub type N4
pub type U9223372036854775808 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
pub type U10000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
pub type P10000 = PInt<U10000>; pub type N10000 = NInt<U10000>;
pub type U100000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
pub type P100000 = PInt<U100000>; pub type N100000 = NInt<U100000>;
pub type U1000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
```

```
    pub type P1000000 = PInt<U1000000>; pub type N1000000 = NInt<U1000000>;
    pub type U10000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P10000000 = PInt<U10000000>; pub type N10000000 = NInt<U100000
    pub type U100000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P100000000 = PInt<U100000000>; pub type N100000000 = NInt<U100
    pub type U1000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P1000000000 = PInt<U1000000000>; pub type N1000000000 = NInt<U
    pub type U10000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P10000000000 = PInt<U10000000000>; pub type N10000000000 = NIn
    pub type U100000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P100000000000 = PInt<U100000000000>; pub type N100000000000 =
    pub type U1000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P1000000000000 = PInt<U1000000000000>; pub type N1000000000000
    pub type U10000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P10000000000000 = PInt<U10000000000000>; pub type N10000000000
    pub type U100000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
    pub type P100000000000000 = PInt<U100000000000000>; pub type N100000000
    pub type U1000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UI
    pub type P1000000000000000 = PInt<U1000000000000000>; pub type N1000000
    pub type U10000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<U
    pub type P10000000000000000 = PInt<U10000000000000000>; pub type N10000
    pub type U100000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<
    pub type P100000000000000000 = PInt<U100000000000000000>; pub type N100
    pub type U1000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt
    pub type P1000000000000000000 = PInt<U1000000000000000000>; pub type N1
    pub type U10000000000000000000 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn
}
```

File: ./target/debug/build/typenum-0568521468b798f9/out/tests.rs

```
extern crate typenum;

use std::ops::*;
use std::cmp::Ordering;
use typenum::*;

#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_BitOr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitOrU0 = <<A as BitOr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitOrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitXorU0 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitXorU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU0 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU0 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_0() {
    type A = UTerm;
    type B = UTerm;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0AddU0 = <<A as Add<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0AddU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MaxU0 = <<A as Max<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MaxU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0GcdU0 = <<A as Gcd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0GcdU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Sub_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0SubU0 = <<A as Sub<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0SubU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_0() {
    type A = UTerm;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_0() {
    type A = UTerm;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_0() {
    type A = UTerm;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U0CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_1() {
```

```
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU1 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0BitXorU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU1 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U0AddU1 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0AddU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU1 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U0GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU1 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU1 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_1() {
    type A = UTerm;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU1 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_1() {
```

```
    type A = UTerm;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU2 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0BitXorU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU2 = <<A as Shl<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U0ShlU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU2 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0AddU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU2 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_Gcd_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U0GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU2 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU2 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_2() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitXorU3 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0BitXorU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU3 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU3 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0AddU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_3() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU3 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U0GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU3 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U0DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU3 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_3() {
    type A = UTerm;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_BitAnd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0BitXorU4 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0BitXorU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU4 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0AddU4 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0AddU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU4 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U0MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U0GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU4 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_0_Pow_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU4 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_4() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U0CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitAnd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitOr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_0_BitXor_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U0BitXorU5 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0BitXorU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shl_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShlU5 = <<A as Shl<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShlU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Shr_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Add_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0AddU5 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0AddU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Min_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MinU5 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MinU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_0_Max_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Gcd_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U0GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Mul_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0MulU5 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0MulU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Div_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Rem_5() {
```

```
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_PartialDiv_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U0>>::Outpu

    assert_eq!(<U0PartialDivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_0_Pow_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U0PowU5 = <<A as Pow<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U0PowU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_0_Cmp_5() {
    type A = UTerm;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U0CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U0CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U1BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU0 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU0 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitXorU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShlU0 = <<A as Shl<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShlU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1ShrU0 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1ShrU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Add_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU0 = <<A as Add<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1AddU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU0 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU0 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1SubU0 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1SubU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_0() {
    type A = UInt<UTerm, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U1CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU1 = <<A as BitOr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitOrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitXorU1 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitXorU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU1 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1ShlU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU1 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_1() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU1 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1AddU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU1 = <<A as Max<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MaxU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Sub_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U1SubU1 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1SubU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU1 = <<A as Mul<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MulU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1DivU1 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1DivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_PartialDiv_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U1PartialDivU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU1 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_1() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU2 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitXorU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU2 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1ShlU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU2 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1AddU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU2 = <<A as Min<B>>::Output as Same<U1>>::Output;
```

```
    assert_eq!(<U1MinU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU2 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1MulU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU2 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Rem_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU2 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_2() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U1BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU3 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U1BitXorU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU3 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U1ShlU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU3 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1AddU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU3 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU3 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U1MulU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_3() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU3 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_3() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;
```

```
    assert_eq!(<U1BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitXorU4 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitXorU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1ShlU4 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U1ShlU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_1_Add_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1AddU4 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1AddU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU4 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1MulU4 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1MulU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU4 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_4() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU4 as Ord>::to_ordering(), Ordering::Less);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitAnd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitOr_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_1_BitXor_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U1BitXorU5 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U1BitXorU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shl_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U1ShlU5 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U1ShlU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_1_Shr_5() {
```

```
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Add_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U1AddU5 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U1AddU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Min_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1MinU5 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1MinU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Max_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Gcd_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U1GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Mul_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1MulU5 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U1MulU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Div_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U1DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U1DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Rem_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1RemU5 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1RemU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_1_Pow_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U1PowU5 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U1PowU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_1_Cmp_5() {
    type A = UInt<UTerm, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U1CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U1CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU0 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU0 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitXorU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
```

```
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU0 = <<A as Shl<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShlU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShrU0 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2ShrU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU0 = <<A as Add<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2AddU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU0 = <<A as Max<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2MaxU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU0 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2SubU0 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2SubU0 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64());
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U2CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU1 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitXorU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2ShlU1 = <<A as Shl<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2ShlU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU1 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2AddU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU1 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2SubU1 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2SubU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU1 = <<A as Mul<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MulU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2DivU1 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2DivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_1() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U2PartialDivU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU1 = <<A as Pow<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2PowU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U2BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU2 = <<A as BitOr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitOrU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitXorU2 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitXorU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU2 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2ShlU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU2 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2AddU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU2 = <<A as Max<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MaxU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2SubU2 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2SubU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU2 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MulU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_PartialDiv_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu
```

```
    assert_eq!(<U2PartialDivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU2 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2PowU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitAndU3 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2BitAndU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_3() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU3 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2BitXorU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U16 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2ShlU3 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2ShlU3 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU3 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2AddU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U2MinU3 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U2MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU3 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2MulU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU3 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU3 = <<A as Pow<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2PowU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitOrU4 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitOrU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2BitXorU4 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U2BitXorU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU4 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2ShlU4 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2AddU4 = <<A as Add<B>>::Output as Same<U6>>::Output;
```

```
    assert_eq!(<U2AddU4 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU4 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U2MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2GcdU4 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2GcdU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU4 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U2MulU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_2_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU4 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU4 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2PowU4 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U2PowU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U2CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitAnd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U2BitAndU5 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2BitAndU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_2_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2BitXorU5 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2BitXorU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2ShlU5 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U2ShlU5 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_2_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U2AddU5 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U2AddU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MinU5 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2MinU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U2MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U2MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U2GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U2GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Mul_5() {
```

```
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2MulU5 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U2MulU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U2DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U2DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U2RemU5 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U2RemU5 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_2_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U2PowU5 = <<A as Pow<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U2PowU5 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_2_Cmp_5() {
    type A = UInt<UInt<UTerm, B1>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
```

```
    type U2CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U2CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU0 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU0 = <<A as BitXor<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitXorU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShlU0 = <<A as Shl<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShlU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_Shr_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU0 = <<A as Shr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3ShrU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU0 = <<A as Add<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3AddU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU0 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
```

```
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU0 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU0 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3SubU0 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_0() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U3CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU0 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU1 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU1 = <<A as BitXor<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitXorU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU1 = <<A as Shl<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3ShlU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_1() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3ShrU1 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3ShrU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU1 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3AddU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU1 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U3GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3SubU1 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3SubU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU1 = <<A as Mul<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MulU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU1 = <<A as Div<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3DivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U3>>::Outpu

    assert_eq!(<U3PartialDivU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU1 = <<A as Pow<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3PowU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_1() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitAndU2 = <<A as BitAnd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3BitAndU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU2 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU2 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitXorU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU2 = <<A as Shl<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U3ShlU2 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU2 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU2 = <<A as Add<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U3AddU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U3MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU2 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3SubU2 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3SubU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_Mul_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU2 = <<A as Mul<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3MulU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU2 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U9 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU2 = <<A as Pow<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3PowU2 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_2() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U3CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU3 = <<A as BitAnd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitAndU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU3 = <<A as BitOr<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3BitOrU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitXorU3 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitXorU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U24 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3ShlU3 = <<A as Shl<B>>::Output as Same<U24>>::Output;

    assert_eq!(<U3ShlU3 as Unsigned>::to_u64(), <U24 as Unsigned>::to_u64()
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU3 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3AddU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU3 = <<A as Max<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MaxU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_3() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU3 = <<A as Gcd<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3GcdU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Sub_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3SubU3 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3SubU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU3 = <<A as Mul<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U3MulU3 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U3RemU3 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3RemU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_PartialDiv_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3PartialDivU3 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U3PartialDivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U27 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3PowU3 = <<A as Pow<B>>::Output as Same<U27>>::Output;

    assert_eq!(<U3PowU3 as Unsigned>::to_u64(), <U27 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_3() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3BitAndU4 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3BitAndU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_3_BitOr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU4 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitXorU4 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitXorU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U48 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU4 = <<A as Shl<B>>::Output as Same<U48>>::Output;

    assert_eq!(<U3ShlU4 as Unsigned>::to_u64(), <U48 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3AddU4 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3AddU4 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU4 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U3MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3MulU4 = <<A as Mul<B>>::Output as Same<U12>>::Output;
```

```
    assert_eq!(<U3MulU4 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU4 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU4 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU4 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U81 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U3PowU4 = <<A as Pow<B>>::Output as Same<U81>>::Output;

    assert_eq!(<U3PowU4 as Unsigned>::to_u64(), <U81 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_4() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitAnd_5() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3BitAndU5 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3BitAndU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitOr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3BitOrU5 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U3BitOrU5 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_BitXor_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U3BitXorU5 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U3BitXorU5 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shl_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U96 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U3ShlU5 = <<A as Shl<B>>::Output as Same<U96>>::Output;

    assert_eq!(<U3ShlU5 as Unsigned>::to_u64(), <U96 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Shr_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U3ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Add_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U3AddU5 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U3AddU5 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Min_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MinU5 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3MinU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Max_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U3MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Gcd_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U3GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U3GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_3_Mul_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U15 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3MulU5 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U3MulU5 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_3_Div_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U3DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U3DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Rem_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U3RemU5 = <<A as Rem<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U3RemU5 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_3_Pow_5() {
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U243 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U3PowU5 = <<A as Pow<B>>::Output as Same<U243>>::Output;

    assert_eq!(<U3PowU5 as Unsigned>::to_u64(), <U243 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_3_Cmp_5() {
```

```
    type A = UInt<UInt<UTerm, B1>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U3CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U3CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU0 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU0 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitXorU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU0 = <<A as Shl<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4ShlU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU0 = <<A as Shr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4ShrU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU0 = <<A as Add<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4AddU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU0 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU0 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU0 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4SubU0 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU1 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU1 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitXorU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU1 = <<A as Shl<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4ShlU1 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU1 = <<A as Add<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4AddU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU1 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_1() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU1 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4SubU1 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU1 = <<A as Mul<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MulU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU1 = <<A as Div<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4DivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;
```

```
    #[allow(non_camel_case_types)]
    type U4RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U4>>::Outpu

    assert_eq!(<U4PartialDivU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU1 = <<A as Pow<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4PowU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU2 = <<A as BitOr<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitOrU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitXorU2 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4BitXorU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4ShlU2 = <<A as Shl<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4ShlU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU2 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U4AddU2 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU2 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU2 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU2 = <<A as Gcd<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4GcdU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4SubU2 = <<A as Sub<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U4SubU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU2 = <<A as Mul<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4MulU2 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U4DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU2 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU2 = <<A as PartialDiv<B>>::Output as Same<U2>>::Outpu

    assert_eq!(<U4PartialDivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4PowU2 = <<A as Pow<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4PowU2 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitAndU3 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitAndU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U4BitXorU3 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4BitXorU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U32 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU3 = <<A as Shl<B>>::Output as Same<U32>>::Output;

    assert_eq!(<U4ShlU3 as Unsigned>::to_u64(), <U32 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU3 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U4AddU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U4MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU3 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU3 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4SubU3 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4SubU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U12 = UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU3 = <<A as Mul<B>>::Output as Same<U12>>::Output;

    assert_eq!(<U4MulU3 as Unsigned>::to_u64(), <U12 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_3() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4RemU3 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4RemU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4PowU3 = <<A as Pow<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4PowU3 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;
```

```
    assert_eq!(<U4BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitOrU4 = <<A as BitOr<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitOrU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4BitXorU4 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4BitXorU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U64 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U4ShlU4 = <<A as Shl<B>>::Output as Same<U64>>::Output;

    assert_eq!(<U4ShlU4 as Unsigned>::to_u64(), <U64 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_4_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4AddU4 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U4AddU4 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MaxU4 = <<A as Max<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MaxU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4GcdU4 = <<A as Gcd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4GcdU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4SubU4 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4SubU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U16 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU4 = <<A as Mul<B>>::Output as Same<U16>>::Output;

    assert_eq!(<U4MulU4 as Unsigned>::to_u64(), <U16 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4RemU4 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4RemU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_PartialDiv_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4PartialDivU4 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu
```

```
    assert_eq!(<U4PartialDivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U256 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0

    #[allow(non_camel_case_types)]
    type U4PowU4 = <<A as Pow<B>>::Output as Same<U256>>::Output;

    assert_eq!(<U4PowU4 as Unsigned>::to_u64(), <U256 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4BitAndU5 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4BitAndU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_BitXor_5() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4BitXorU5 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4BitXorU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U128 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0

    #[allow(non_camel_case_types)]
    type U4ShlU5 = <<A as Shl<B>>::Output as Same<U128>>::Output;

    assert_eq!(<U4ShlU5 as Unsigned>::to_u64(), <U128 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_4_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4AddU5 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U4AddU5 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Min_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U4MinU5 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4MinU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U4MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U4GcdU5 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U4GcdU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4MulU5 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U4MulU5 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_4_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U4DivU5 = <<A as Div<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U4DivU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_4_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U4RemU5 = <<A as Rem<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U4RemU5 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_4_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1024 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTe

    #[allow(non_camel_case_types)]
    type U4PowU5 = <<A as Pow<B>>::Output as Same<U1024>>::Output;

    assert_eq!(<U4PowU5 as Unsigned>::to_u64(), <U1024 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_4_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U4CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U4CmpU5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU0 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
```

```
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU0 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU0 = <<A as BitXor<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitXorU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShlU0 = <<A as Shl<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShlU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU0 = <<A as Shr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5ShrU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU0 = <<A as Add<B>>::Output as Same<U5>>::Output;
```

```
    assert_eq!(<U5AddU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MinU0 = <<A as Min<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MinU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU0 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU0 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU0 = <<A as Sub<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5SubU0 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Mul_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5MulU0 = <<A as Mul<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5MulU0 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU0 = <<A as Pow<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5PowU0 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_0() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UTerm;

    #[allow(non_camel_case_types)]
    type U5CmpU0 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU1 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5BitOrU1 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU1 = <<A as BitXor<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitXorU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU1 = <<A as Shl<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5ShlU1 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShrU1 = <<A as Shr<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5ShrU1 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU1 = <<A as Add<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5AddU1 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU1 = <<A as Min<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5MinU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU1 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU1 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU1 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU1 = <<A as Sub<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5SubU1 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_1() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU1 = <<A as Mul<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MulU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU1 = <<A as Div<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5DivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU1 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU1 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU1 = <<A as PartialDiv<B>>::Output as Same<U5>>::Outpu

    assert_eq!(<U5PartialDivU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5PowU1 = <<A as Pow<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5PowU1 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_1() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU1 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitAndU2 = <<A as BitAnd<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitAndU2 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU2 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU2 = <<A as BitXor<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitXorU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Shl_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5ShlU2 = <<A as Shl<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5ShlU2 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5ShrU2 = <<A as Shr<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5ShrU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU2 = <<A as Add<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5AddU2 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU2 = <<A as Min<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5MinU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
```

```
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU2 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU2 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU2 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU2 = <<A as Sub<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5SubU2 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU2 = <<A as Mul<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5MulU2 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5DivU2 = <<A as Div<B>>::Output as Same<U2>>::Output;
```

```
    assert_eq!(<U5DivU2 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU2 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU2 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5PowU2 = <<A as Pow<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5PowU2 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_2() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU2 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU3 = <<A as BitAnd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitAndU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_3() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U7 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU3 = <<A as BitOr<B>>::Output as Same<U7>>::Output;

    assert_eq!(<U5BitOrU3 as Unsigned>::to_u64(), <U7 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U6 = UInt<UInt<UInt<UTerm, B1>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitXorU3 = <<A as BitXor<B>>::Output as Same<U6>>::Output;

    assert_eq!(<U5BitXorU3 as Unsigned>::to_u64(), <U6 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U40 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU3 = <<A as Shl<B>>::Output as Same<U40>>::Output;

    assert_eq!(<U5ShlU3 as Unsigned>::to_u64(), <U40 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU3 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU3 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U8 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>;
```

```
    #[allow(non_camel_case_types)]
    type U5AddU3 = <<A as Add<B>>::Output as Same<U8>>::Output;

    assert_eq!(<U5AddU3 as Unsigned>::to_u64(), <U8 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U3 = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU3 = <<A as Min<B>>::Output as Same<U3>>::Output;

    assert_eq!(<U5MinU3 as Unsigned>::to_u64(), <U3 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU3 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU3 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU3 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5SubU3 = <<A as Sub<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5SubU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U15 = UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5MulU3 = <<A as Mul<B>>::Output as Same<U15>>::Output;

    assert_eq!(<U5MulU3 as Unsigned>::to_u64(), <U15 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU3 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU3 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U2 = UInt<UInt<UTerm, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5RemU3 = <<A as Rem<B>>::Output as Same<U2>>::Output;

    assert_eq!(<U5RemU3 as Unsigned>::to_u64(), <U2 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_3() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;
    type U125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type U5PowU3 = <<A as Pow<B>>::Output as Same<U125>>::Output;

    assert_eq!(<U5PowU3 as Unsigned>::to_u64(), <U125 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_3() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UTerm, B1>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU3 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5BitAndU4 = <<A as BitAnd<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5BitAndU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU4 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5BitXorU4 = <<A as BitXor<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5BitXorU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U80 = UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>

    #[allow(non_camel_case_types)]
    type U5ShlU4 = <<A as Shl<B>>::Output as Same<U80>>::Output;
```

```
    assert_eq!(<U5ShlU4 as Unsigned>::to_u64(), <U80 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU4 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU4 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U9 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5AddU4 = <<A as Add<B>>::Output as Same<U9>>::Output;

    assert_eq!(<U5AddU4 as Unsigned>::to_u64(), <U9 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U4 = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MinU4 = <<A as Min<B>>::Output as Same<U4>>::Output;

    assert_eq!(<U5MinU4 as Unsigned>::to_u64(), <U4 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU4 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU4 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_5_Gcd_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU4 = <<A as Gcd<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5GcdU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5SubU4 = <<A as Sub<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5SubU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U20 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5MulU4 = <<A as Mul<B>>::Output as Same<U20>>::Output;

    assert_eq!(<U5MulU4 as Unsigned>::to_u64(), <U20 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU4 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
```

```
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5RemU4 = <<A as Rem<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5RemU4 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;
    type U625 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type U5PowU4 = <<A as Pow<B>>::Output as Same<U625>>::Output;

    assert_eq!(<U5PowU4 as Unsigned>::to_u64(), <U625 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_4() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B0>;

    #[allow(non_camel_case_types)]
    type U5CmpU4 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitAnd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitAndU5 = <<A as BitAnd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitAndU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitOr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5BitOrU5 = <<A as BitOr<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5BitOrU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64(
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_BitXor_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5BitXorU5 = <<A as BitXor<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5BitXorU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shl_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U160 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1

    #[allow(non_camel_case_types)]
    type U5ShlU5 = <<A as Shl<B>>::Output as Same<U160>>::Output;

    assert_eq!(<U5ShlU5 as Unsigned>::to_u64(), <U160 as Unsigned>::to_u64(
}
#[test]
#[allow(non_snake_case)]
fn test_5_Shr_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5ShrU5 = <<A as Shr<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5ShrU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Add_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U10 = UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>;

    #[allow(non_camel_case_types)]
    type U5AddU5 = <<A as Add<B>>::Output as Same<U10>>::Output;

    assert_eq!(<U5AddU5 as Unsigned>::to_u64(), <U10 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Min_5() {
```

```
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MinU5 = <<A as Min<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MinU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Max_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5MaxU5 = <<A as Max<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5MaxU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Gcd_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U5 = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5GcdU5 = <<A as Gcd<B>>::Output as Same<U5>>::Output;

    assert_eq!(<U5GcdU5 as Unsigned>::to_u64(), <U5 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Sub_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5SubU5 = <<A as Sub<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5SubU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Mul_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U25 = UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>;
```

```
    #[allow(non_camel_case_types)]
    type U5MulU5 = <<A as Mul<B>>::Output as Same<U25>>::Output;

    assert_eq!(<U5MulU5 as Unsigned>::to_u64(), <U25 as Unsigned>::to_u64()
}
#[test]
#[allow(non_snake_case)]
fn test_5_Div_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5DivU5 = <<A as Div<B>>::Output as Same<U1>>::Output;

    assert_eq!(<U5DivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_Rem_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U0 = UTerm;

    #[allow(non_camel_case_types)]
    type U5RemU5 = <<A as Rem<B>>::Output as Same<U0>>::Output;

    assert_eq!(<U5RemU5 as Unsigned>::to_u64(), <U0 as Unsigned>::to_u64())
}
#[test]
#[allow(non_snake_case)]
fn test_5_PartialDiv_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U1 = UInt<UTerm, B1>;

    #[allow(non_camel_case_types)]
    type U5PartialDivU5 = <<A as PartialDiv<B>>::Output as Same<U1>>::Outpu

    assert_eq!(<U5PartialDivU5 as Unsigned>::to_u64(), <U1 as Unsigned>::to
}
#[test]
#[allow(non_snake_case)]
fn test_5_Pow_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type U3125 = UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type U5PowU5 = <<A as Pow<B>>::Output as Same<U3125>>::Output;

    assert_eq!(<U5PowU5 as Unsigned>::to_u64(), <U3125 as Unsigned>::to_u64
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_5_Cmp_5() {
    type A = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;
    type B = UInt<UInt<UInt<UTerm, B1>, B0>, B1>;

    #[allow(non_camel_case_types)]
    type U5CmpU5 = <A as Cmp<B>>::Output;
    assert_eq!(<U5CmpU5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN5 = <<A as Add<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5AddN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5SubN5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5SubN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulN5 = <<A as Mul<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5MulN5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN5 = <<A as Max<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MaxN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N5PartialDivN5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN4 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5AddN4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5SubN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N4() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulN4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N5MulN4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN3 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5AddN3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5SubN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N5MulN3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddN2 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5AddN2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubN2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5SubN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulN2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N5MulN2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_N2() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddN1 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5AddN1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubN1 = <<A as Sub<B>>::Output as Same<N4>>::Output;
```

```
    assert_eq!(<N5SubN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulN1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MulN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinN1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivN1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5DivN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<N5PartialDivN1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5Add_0 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Add_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Sub_0 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Sub_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Min_0 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5Min_0 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N5Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N5AddP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP1 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N5SubP1 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MulP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP1 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5DivP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu

    assert_eq!(<N5PartialDivP1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5PowP1 = <<A as Pow<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5PowP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N5AddP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP2 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N5SubP2 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MulP2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5MulP2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5MinP2 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N5MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<N5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5AddP3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5AddP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP3 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N5SubP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MulP3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N5MulP3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP3 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N5MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N5DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N5RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1
```

```
    #[allow(non_camel_case_types)]
    type N5PowP3 = <<A as Pow<B>>::Output as Same<N125>>::Output;

    assert_eq!(<N5PowP3 as Integer>::to_i64(), <N125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5AddP4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5AddP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P4() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5SubP4 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N5SubP4 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N5MulP4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N5MulP4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP4 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N5MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type N5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<N5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N5_Add_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5AddP5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5AddP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N5SubP5 = <<A as Sub<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N5SubP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N5MulP5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<N5MulP5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MinP5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N5MinP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5DivP5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N5DivP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_PartialDiv_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu
```

```
    assert_eq!(<N5PartialDivP5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3125 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N5PowP5 = <<A as Pow<B>>::Output as Same<N3125>>::Output;

    assert_eq!(<N5PowP5 as Integer>::to_i64(), <N3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N5CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN5 = <<A as Add<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4AddN5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN5 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4SubN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N5() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<N4MulN5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN5 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemN5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN4 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4AddN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4SubN4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4SubN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Mul_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulN4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4MulN4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN4 = <<A as Max<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MaxN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N4PartialDivN4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN3 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4AddN3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4SubN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N4MulN3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N3() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddN2 = <<A as Add<B>>::Output as Same<N6>>::Output;
```

```
    assert_eq!(<N4AddN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubN2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4SubN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N4MulN2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Gcd_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4DivN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N4PartialDivN2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddN1 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4AddN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubN1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4SubN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulN1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MulN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinN1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivN1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4DivN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_N1() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<N4PartialDivN1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_N1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Add_0 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Add_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Sub_0 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Sub_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Min_0 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4Min_0 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Cmp__0() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N4Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N4AddP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP1 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N4SubP1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MulP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MinP1 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4DivP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<N4PartialDivP1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PowP1 = <<A as Pow<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4PowP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P1() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4AddP2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4AddP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP2 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N4SubP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4MulP2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP2 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N4MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;
```

```
    assert_eq!(<N4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4DivP2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N4DivP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N4PartialDivP2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N4_Cmp_P2() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4AddP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP3 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N4SubP3 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MulP3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N4MulP3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MinP3 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N4MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N4RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N64 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4PowP3 = <<A as Pow<B>>::Output as Same<N64>>::Output;

    assert_eq!(<N4PowP3 as Integer>::to_i64(), <N64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P3() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4AddP4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4AddP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4SubP4 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N4SubP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N16 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<N4MulP4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4DivP4 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N4DivP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_PartialDiv_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N4PartialDivP4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type N4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<N4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P4() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Add_P5() {
```

```
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4AddP5 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4AddP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Sub_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4SubP5 = <<A as Sub<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N4SubP5 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Mul_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N4MulP5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<N4MulP5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Min_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4MinP5 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4MinP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Max_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Gcd_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Div_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Rem_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N4RemP5 = <<A as Rem<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N4RemP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Pow_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1024 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type N4PowP5 = <<A as Pow<B>>::Output as Same<N1024>>::Output;

    assert_eq!(<N4PowP5 as Integer>::to_i64(), <N1024 as Integer>::to_i64()
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Cmp_P5() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN5 = <<A as Add<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3AddN5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN5 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3SubN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<N3MulN5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN5 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN5 = <<A as Rem<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3RemN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN4 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3AddN4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3SubN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<N3MulN4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N4() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN4 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3RemN4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN3 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3AddN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3SubN3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3SubN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3MulN3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN3 = <<A as Max<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MaxN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N3PartialDivN3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddN2 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3AddN2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubN2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3SubN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulN2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N3MulN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_N2() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddN1 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3AddN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubN1 = <<A as Sub<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N3SubN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulN1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MulN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinN1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivN1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3DivN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<N3PartialDivN1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3Add_0 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Add_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Sub_0 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Sub_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Min_0 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3Min_0 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N3Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP1 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N3AddP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP1 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N3SubP1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MulP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP1 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<N3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3DivP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu

    assert_eq!(<N3PartialDivP1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP1 = <<A as Pow<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3PowP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N3_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3AddP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP2 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N3SubP2 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3MulP2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MinP2 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N3RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<N3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3AddP3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3AddP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP3 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N3SubP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type N9 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<N3MulP3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3DivP3 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N3DivP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_PartialDiv_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N3PartialDivP3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N27 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP3 = <<A as Pow<B>>::Output as Same<N27>>::Output;

    assert_eq!(<N3PowP3 as Integer>::to_i64(), <N27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P4() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3AddP4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3AddP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3SubP4 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N3SubP4 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3MulP4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<N3MulP4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP4 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type N3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP4 = <<A as Rem<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3RemP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type N3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<N3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N3AddP5 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N3AddP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N3SubP5 = <<A as Sub<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N3SubP5 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Mul_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MulP5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<N3MulP5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MinP5 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N3MinP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N3RemP5 = <<A as Rem<B>>::Output as Same<N3>>::Output;
```

```
    assert_eq!(<N3RemP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N243 = NInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type N3PowP5 = <<A as Pow<B>>::Output as Same<N243>>::Output;

    assert_eq!(<N3PowP5 as Integer>::to_i64(), <N243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN5 = <<A as Add<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2AddN5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN5 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2SubN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N5() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<N2MulN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN5 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN4 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2AddN4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubN4 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2SubN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Mul_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<N2MulN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN4 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN4 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddN3 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2AddN3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2SubN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<N2MulN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN3 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N3() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemN3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddN2 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2AddN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2SubN2 = <<A as Sub<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N2SubN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MulN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxN2 = <<A as Max<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MaxN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N2_Div_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivN2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2DivN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N2PartialDivN2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N2AddN1 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2AddN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubN1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2SubN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulN1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MulN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinN1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivN1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2DivN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_N1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<N2PartialDivN1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_N1() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpN1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Add_0 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Add_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Sub_0 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2Sub_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Min_0 = <<A as Min<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N2Min_0 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp__0() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N2Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P1() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N2AddP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP1 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N2SubP1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MulP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP1 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2DivP1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2DivP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<N2PartialDivP1 as Integer>::to_i64(), <N2 as Integer>::to_i
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP1 = <<A as Pow<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2PowP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P1() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2AddP2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2AddP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP2 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2SubP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N2MulP2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2DivP2 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N2DivP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_PartialDiv_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N2PartialDivP2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P2() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P3() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2AddP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP3 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N2SubP3 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2MulP3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP3 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP3 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N8 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2PowP3 = <<A as Pow<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2PowP3 as Integer>::to_i64(), <N8 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P3() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2AddP4 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2AddP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2SubP4 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N2SubP4 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP4 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<N2MulP4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP4 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP4 = <<A as Rem<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N2RemP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<N2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P4() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Add_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2AddP5 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N2AddP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Sub_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N7 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N2SubP5 = <<A as Sub<B>>::Output as Same<N7>>::Output;

    assert_eq!(<N2SubP5 as Integer>::to_i64(), <N7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Mul_P5() {
```

```
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MulP5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<N2MulP5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Min_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2MinP5 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2MinP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Max_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Gcd_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Div_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type N2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Rem_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N2RemP5 = <<A as Rem<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N2RemP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Pow_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N32 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type N2PowP5 = <<A as Pow<B>>::Output as Same<N32>>::Output;

    assert_eq!(<N2PowP5 as Integer>::to_i64(), <N32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Cmp_P5() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN5 = <<A as Add<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1AddN5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Sub_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN5 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1SubN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MulN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN5 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN4 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1AddN4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN4 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1SubN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MulN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N4() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN4 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN3 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1AddN3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubN3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1SubN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1MulN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Min_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN3 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddN2 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1AddN2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubN2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1SubN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulN2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MulN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN2 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N2() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemN2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddN1 = <<A as Add<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<N1AddN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1SubN1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1SubN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulN1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MulN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxN1 = <<A as Max<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MaxN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Gcd_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivN1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1DivN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_PartialDiv_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<N1PartialDivN1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowN1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_N1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpN1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Add_0 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Add_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Sub_0 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Sub_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Min_0 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1Min_0 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow__0() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp__0() {
```

```
    type A = NInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type N1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<N1Cmp_0 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1AddP1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1AddP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP1 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1SubP1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MulP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP1 = <<A as Min<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<N1MinP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1DivP1 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1DivP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_PartialDiv_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<N1PartialDivP1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP1 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P1() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1AddP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1SubP2 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1SubP2 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<N1MulP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP2 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP2 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P2() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<N1AddP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP3 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1SubP3 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<N1MulP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP3 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<N1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP3 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP3 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_N1_Cmp_P3() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type N1AddP4 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<N1AddP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1SubP4 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1SubP4 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MulP4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<N1MulP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type N1MinP4 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<N1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP4 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P4() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Add_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type N1AddP5 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<N1AddP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Sub_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type N1SubP5 = <<A as Sub<B>>::Output as Same<N6>>::Output;

    assert_eq!(<N1SubP5 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Mul_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MulP5 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<N1MulP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Min_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1MinP5 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1MinP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Max_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<N1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Gcd_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<N1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Div_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type N1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<N1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Rem_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1RemP5 = <<A as Rem<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1RemP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Pow_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type N1PowP5 = <<A as Pow<B>>::Output as Same<N1>>::Output;

    assert_eq!(<N1PowP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Cmp_P5() {
    type A = NInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type N1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<N1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN5 = <<A as Add<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0AddN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N5() {
```

```
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN5 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0SubN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN5 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type _0GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N5() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Add_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN4 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0AddN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN4 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0SubN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN4 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN4 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu
```

```
    assert_eq!(<_0PartialDivN4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N4() {
    type A = Z0;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN3 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0AddN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN3 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0SubN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N3() {
```

```
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN3 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;
```

```
    #[allow(non_camel_case_types)]
    type _0RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N3() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddN2 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0AddN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubN2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0SubN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Mul_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN2 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN2 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N2() {
    type A = Z0;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0AddN1 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0AddN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubN1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0SubN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulN1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MaxN1 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MaxN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_N1() {
```

```
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivN1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivN1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_N1() {
    type A = Z0;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type _0CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Add_0 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Add_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Sub_0 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Sub_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Max__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Max_0 = <<A as Max<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Max_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd__0() {
    type A = Z0;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0Gcd_0 = <<A as Gcd<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0Gcd_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow__0() {
    type A = Z0;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp__0() {
    type A = Z0;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type _0Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<_0Cmp_0 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type _0AddP1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0AddP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP1 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<_0SubP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP1 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP1 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<_0GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP1 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP1 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P1() {
```

```
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP1 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P1() {
    type A = Z0;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP1 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0AddP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP2 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<_0SubP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP2 = <<A as Mul<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MulP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP2 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<_0GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Rem_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP2 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP2 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P2() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type _0AddP3 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0AddP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP3 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<_0SubP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP3 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP3 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<_0GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP3 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P3() {
```

```
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP3 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P3() {
    type A = Z0;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0AddP4 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0AddP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0SubP4 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<_0SubP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP4 = <<A as Mul<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<_0MulP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP4 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<_0GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Rem_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP4 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP4 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P4() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type _0CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test__0_Add_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type _0AddP5 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0AddP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Sub_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0SubP5 = <<A as Sub<B>>::Output as Same<N5>>::Output;

    assert_eq!(<_0SubP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Mul_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MulP5 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MulP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Min_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0MinP5 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0MinP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Max_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test__0_Gcd_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<_0GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Div_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Rem_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_PartialDiv_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<_0>>::Outpu

    assert_eq!(<_0PartialDivP5 as Integer>::to_i64(), <_0 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test__0_Pow_P5() {
```

```
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type _0PowP5 = <<A as Pow<B>>::Output as Same<_0>>::Output;

    assert_eq!(<_0PowP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Cmp_P5() {
    type A = Z0;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type _0CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<_0CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN5 = <<A as Add<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1AddN5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN5 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1SubN5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN5 = <<A as Mul<B>>::Output as Same<N5>>::Output;
```

```
    assert_eq!(<P1MulN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P1MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN5 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Rem_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN4 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1AddN4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1SubN4 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1SubN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN4 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MulN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN4 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddN3 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1AddN3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN3 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1SubN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN3 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MulN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN3 = <<A as Max<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1MaxN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Cmp_N3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddN2 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1AddN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubN2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1SubN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulN2 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MulN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P1MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN2 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivN2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1AddN1 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1AddN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubN1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1SubN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulN1 = <<A as Mul<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MulN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxN1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivN1 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P1DivN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P1PartialDivN1 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowN1 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_N1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add__0() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Add_0 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Add_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Sub_0 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Sub_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1Max_0 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Max_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Gcd_0 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Gcd_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp__0() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P1Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P1Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1AddP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Sub_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1SubP1 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1SubP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP1 = <<A as Mul<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MulP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP1 = <<A as Max<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MaxP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1DivP1 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1DivP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_PartialDiv_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P1PartialDivP1 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP1 = <<A as Pow<B>>::Output as Same<P1>>::Output;
```

```
    assert_eq!(<P1PowP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P1() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP1 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1AddP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP2 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P1SubP2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP2 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MulP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P2() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP2 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P1MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP2 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P1RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP2 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P2() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP2 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP3 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1AddP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP3 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P1SubP3 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Mul_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP3 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MulP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP3 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P1MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP3 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P3() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1AddP4 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1AddP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P1SubP4 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P1SubP4 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MulP4 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MulP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP4 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Max_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P1MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P4() {
```

```
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP4 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P4() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
```

```
    type P1CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Add_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P1AddP5 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P1AddP5 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Sub_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P1SubP5 = <<A as Sub<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P1SubP5 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Mul_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MulP5 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MulP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Min_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1MinP5 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1MinP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P1_Max_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P1MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Gcd_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Div_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P1DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P1DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Rem_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1RemP5 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1RemP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Pow_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P1PowP5 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P1PowP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Cmp_P5() {
    type A = PInt<UInt<UTerm, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P1CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P1CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN5 = <<A as Add<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2AddN5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN5 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2SubN5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N10 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN5 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P2MulN5 as Integer>::to_i64(), <N10 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P2MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN5 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N5() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddN4 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2AddN4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN4 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2SubN4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN4 = <<A as Mul<B>>::Output as Same<N8>>::Output;
```

```
    assert_eq!(<P2MulN4 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN4 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN3 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2AddN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN3 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2SubN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2MulN3 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P2MulN3 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN3 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivN3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2AddN2 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2AddN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubN2 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2SubN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN2 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P2MulN2 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;
```

```
    assert_eq!(<P2DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P2PartialDivN2 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddN1 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2AddN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_N1() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubN1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2SubN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulN1 = <<A as Mul<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2MulN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxN1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P2GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivN1 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2DivN1 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P2PartialDivN1 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Add_0 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Add_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Sub_0 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Sub_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Max_0 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Max_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2Gcd_0 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2Gcd_0 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P2Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P2Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2AddP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP1 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2SubP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP1 = <<A as Mul<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MulP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP1 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P1() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2DivP1 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2DivP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P2PartialDivP1 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P2PowP1 = <<A as Pow<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2PowP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP2 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2AddP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2SubP2 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2SubP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP2 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MulP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP2 = <<A as Max<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MaxP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P2PartialDivP2 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP2 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2PowP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP2 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP3 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2AddP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP3 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P2SubP3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP3 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2MulP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP3 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P2MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P3() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP3 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2PowP3 = <<A as Pow<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2PowP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P2CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP3 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2AddP4 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P2AddP4 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2SubP4 = <<A as Sub<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P2SubP4 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP4 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P2MulP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP4 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P2_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P2MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2GcdP4 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2GcdP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP4 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP4 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP4 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P2PowP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P2CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2AddP5 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P2AddP5 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P2SubP5 = <<A as Sub<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P2SubP5 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MulP5 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P2MulP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2MinP5 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2MinP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P2MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P2GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P2GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P2DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P2DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Rem_P5() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P2RemP5 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P2RemP5 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P32 = PInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P2PowP5 = <<A as Pow<B>>::Output as Same<P32>>::Output;

    assert_eq!(<P2PowP5 as Integer>::to_i64(), <P32 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P2CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P2CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN5 = <<A as Add<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3AddN5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN5 = <<A as Sub<B>>::Output as Same<P8>>::Output;
```

```
    assert_eq!(<P3SubN5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN5 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P3MulN5 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P3MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN5 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Div_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN4 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3AddN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3SubN4 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3SubN4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN4 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P3MulN4 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P3MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN4 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivN4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemN4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3AddN3 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3AddN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN3 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3SubN3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N9 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN3 = <<A as Mul<B>>::Output as Same<N9>>::Output;

    assert_eq!(<P3MulN3 as Integer>::to_i64(), <N9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;
```

```
    assert_eq!(<P3GcdN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN3 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P3PartialDivN3 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_N2() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddN2 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3AddN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubN2 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3SubN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N6 = NInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulN2 = <<A as Mul<B>>::Output as Same<N6>>::Output;

    assert_eq!(<P3MulN2 as Integer>::to_i64(), <N6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P3MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P3MaxN2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN2 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3DivN2 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Add_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddN1 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3AddN1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubN1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3SubN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulN1 = <<A as Mul<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3MulN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxN1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivN1 = <<A as Div<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P3DivN1 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N3>>::Outpu
```

```
    assert_eq!(<P3PartialDivN1 as Integer>::to_i64(), <N3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_N1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Add_0 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Add_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Sub_0 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Sub_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min__0() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Max_0 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Max_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3Gcd_0 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3Gcd_0 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp__0() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
```

```
    type P3Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P3Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3AddP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP1 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3SubP1 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP1 = <<A as Mul<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MulP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Max_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP1 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP1 = <<A as Div<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3DivP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P3>>::Outpu

    assert_eq!(<P3PartialDivP1 as Integer>::to_i64(), <P3 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP1 = <<A as Pow<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3PowP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P1() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP2 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3AddP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP2 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3SubP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP2 = <<A as Mul<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P3MulP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P3MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP2 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P2() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP2 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3PowP2 = <<A as Pow<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3PowP2 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P2() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP3 = <<A as Add<B>>::Output as Same<P6>>::Output;
```

```
    assert_eq!(<P3AddP3 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3SubP3 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3SubP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP3 = <<A as Mul<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P3MulP3 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP3 = <<A as Max<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MaxP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Gcd_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP3 = <<A as Gcd<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3GcdP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3RemP3 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3RemP3 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_PartialDiv_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3PartialDivP3 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P3PartialDivP3 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P27 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B1>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP3 = <<A as Pow<B>>::Output as Same<P27>>::Output;

    assert_eq!(<P3PowP3 as Integer>::to_i64(), <P27 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P3() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP3 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3AddP4 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P3AddP4 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3SubP4 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P3SubP4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MulP4 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P3MulP4 as Integer>::to_i64(), <P12 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP4 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P3MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Div_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP4 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P4() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP4 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP4 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P81 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P3PowP4 = <<A as Pow<B>>::Output as Same<P81>>::Output;

    assert_eq!(<P3PowP4 as Integer>::to_i64(), <P81 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P4() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP4 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Add_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P3AddP5 = <<A as Add<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P3AddP5 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Sub_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P3SubP5 = <<A as Sub<B>>::Output as Same<N2>>::Output;
```

```
    assert_eq!(<P3SubP5 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Mul_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MulP5 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P3MulP5 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Min_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MinP5 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3MinP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Max_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P3MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Gcd_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P3GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P3GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P3_Div_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P3DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P3DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Rem_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P3RemP5 = <<A as Rem<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P3RemP5 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Pow_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P243 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1

    #[allow(non_camel_case_types)]
    type P3PowP5 = <<A as Pow<B>>::Output as Same<P243>>::Output;

    assert_eq!(<P3PowP5 as Integer>::to_i64(), <P243 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Cmp_P5() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P3CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P3CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P4AddN5 = <<A as Add<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4AddN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN5 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4SubN5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN5 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P4MulN5 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P4MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN5 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivN5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemN5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemN5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4AddN4 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4AddN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN4 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4SubN4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N16 = NInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulN4 = <<A as Mul<B>>::Output as Same<N16>>::Output;

    assert_eq!(<P4MulN4 as Integer>::to_i64(), <N16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN4 = <<A as Max<B>>::Output as Same<P4>>::Output;
```

```
    assert_eq!(<P4MaxN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN4 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P4PartialDivN4 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN3 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4AddN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN3 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4SubN3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N12 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN3 = <<A as Mul<B>>::Output as Same<N12>>::Output;

    assert_eq!(<P4MulN3 as Integer>::to_i64(), <N12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P4MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P4MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemN3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddN2 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4AddN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubN2 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4SubN2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N8 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN2 = <<A as Mul<B>>::Output as Same<N8>>::Output;

    assert_eq!(<P4MulN2 as Integer>::to_i64(), <N8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdN2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P4DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN2 = <<A as Rem<B>>::Output as Same<_0>>::Output;
```

```
    assert_eq!(<P4RemN2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN2 = <<A as PartialDiv<B>>::Output as Same<N2>>::Outpu

    assert_eq!(<P4PartialDivN2 as Integer>::to_i64(), <N2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddN1 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4AddN1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubN1 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4SubN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_N1() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulN1 = <<A as Mul<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4MulN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxN1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    #[allow(non_camel_case_types)]
    type P4DivN1 = <<A as Div<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P4DivN1 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N4>>::Outpu

    assert_eq!(<P4PartialDivN1 as Integer>::to_i64(), <N4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Add_0 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Add_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Sub_0 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Sub_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Max_0 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Max_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
```

```
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4Gcd_0 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4Gcd_0 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
    type P4Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P4Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP1 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4AddP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP1 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4SubP1 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP1 = <<A as Mul<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MulP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP1 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P1() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP1 = <<A as Div<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4DivP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P4>>::Outpu

    assert_eq!(<P4PartialDivP1 as Integer>::to_i64(), <P4 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PowP1 = <<A as Pow<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4PowP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type P4CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP2 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P4AddP2 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4SubP2 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4SubP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP2 = <<A as Mul<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P4MulP2 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP2 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP2 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP2 = <<A as Gcd<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4GcdP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P4DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP2 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP2 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP2 = <<A as PartialDiv<B>>::Output as Same<P2>>::Outpu

    assert_eq!(<P4PartialDivP2 as Integer>::to_i64(), <P2 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP2 = <<A as Pow<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4PowP2 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP3 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P4AddP3 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP3 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4SubP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P12 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MulP3 = <<A as Mul<B>>::Output as Same<P12>>::Output;

    assert_eq!(<P4MulP3 as Integer>::to_i64(), <P12 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P4MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP3 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP3 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P3() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4RemP3 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4RemP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P64 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4PowP3 = <<A as Pow<B>>::Output as Same<P64>>::Output;

    assert_eq!(<P4PowP3 as Integer>::to_i64(), <P64 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4AddP4 = <<A as Add<B>>::Output as Same<P8>>::Output;
```

```
    assert_eq!(<P4AddP4 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4SubP4 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4SubP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P16 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP4 = <<A as Mul<B>>::Output as Same<P16>>::Output;

    assert_eq!(<P4MulP4 as Integer>::to_i64(), <P16 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MaxP4 = <<A as Max<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MaxP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P4_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4GcdP4 = <<A as Gcd<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4GcdP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4RemP4 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4RemP4 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_PartialDiv_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4PartialDivP4 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P4PartialDivP4 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
```

```
    type P256 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1

    #[allow(non_camel_case_types)]
    type P4PowP4 = <<A as Pow<B>>::Output as Same<P256>>::Output;

    assert_eq!(<P4PowP4 as Integer>::to_i64(), <P256 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP4 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4AddP5 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P4AddP5 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4SubP5 = <<A as Sub<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P4SubP5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P4MulP5 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P4MulP5 as Integer>::to_i64(), <P20 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4MinP5 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4MinP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P4MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P4GcdP5 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P4GcdP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P4DivP5 = <<A as Div<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P4DivP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Rem_P5() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P4RemP5 = <<A as Rem<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P4RemP5 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1024 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P4PowP5 = <<A as Pow<B>>::Output as Same<P1024>>::Output;

    assert_eq!(<P4PowP5 as Integer>::to_i64(), <P1024 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P4CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P4CmpP5 as Ord>::to_ordering(), Ordering::Less);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5AddN5 = <<A as Add<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5AddN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN5 = <<A as Sub<B>>::Output as Same<P10>>::Output;
```

```
    assert_eq!(<P5SubN5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N25 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulN5 = <<A as Mul<B>>::Output as Same<N25>>::Output;

    assert_eq!(<P5MulN5 as Integer>::to_i64(), <N25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN5 = <<A as Min<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MinN5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Div_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN5 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN5 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN5 = <<A as PartialDiv<B>>::Output as Same<N1>>::Outpu

    assert_eq!(<P5PartialDivN5 as Integer>::to_i64(), <N1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN5 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5AddN4 = <<A as Add<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5AddN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN4 = <<A as Sub<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5SubN4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N20 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulN4 = <<A as Mul<B>>::Output as Same<N20>>::Output;

    assert_eq!(<P5MulN4 as Integer>::to_i64(), <N20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN4 = <<A as Min<B>>::Output as Same<N4>>::Output;

    assert_eq!(<P5MinN4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN4 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN4 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN3 = <<A as Add<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5AddN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN3 = <<A as Sub<B>>::Output as Same<P8>>::Output;

    assert_eq!(<P5SubN3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N15 = NInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN3 = <<A as Mul<B>>::Output as Same<N15>>::Output;

    assert_eq!(<P5MulN3 as Integer>::to_i64(), <N15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN3 = <<A as Min<B>>::Output as Same<N3>>::Output;

    assert_eq!(<P5MinN3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN3 = <<A as Max<B>>::Output as Same<P5>>::Output;
```

```
    assert_eq!(<P5MaxN3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN3 = <<A as Div<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5DivN3 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemN3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemN3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_N2() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddN2 = <<A as Add<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5AddN2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubN2 = <<A as Sub<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5SubN2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N10 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulN2 = <<A as Mul<B>>::Output as Same<N10>>::Output;

    assert_eq!(<P5MulN2 as Integer>::to_i64(), <N10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinN2 = <<A as Min<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5MinN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5MaxN2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivN2 = <<A as Div<B>>::Output as Same<N2>>::Output;

    assert_eq!(<P5DivN2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemN2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemN2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpN2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Add_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddN1 = <<A as Add<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5AddN1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubN1 = <<A as Sub<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5SubN1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulN1 = <<A as Mul<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5MulN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinN1 = <<A as Min<B>>::Output as Same<N1>>::Output;

    assert_eq!(<P5MinN1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
```

```
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxN1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxN1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdN1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivN1 = <<A as Div<B>>::Output as Same<N5>>::Output;

    assert_eq!(<P5DivN1 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemN1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemN1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivN1 = <<A as PartialDiv<B>>::Output as Same<N5>>::Outpu
```

```
    assert_eq!(<P5PartialDivN1 as Integer>::to_i64(), <N5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_N1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpN1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpN1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Add_0 = <<A as Add<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Add_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Sub_0 = <<A as Sub<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Sub_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Mul_0 = <<A as Mul<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Mul_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min__0() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5Min_0 = <<A as Min<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5Min_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Max_0 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Max_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5Gcd_0 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5Gcd_0 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5Pow_0 = <<A as Pow<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5Pow_0 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp__0() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = Z0;

    #[allow(non_camel_case_types)]
```

```
    type P5Cmp_0 = <A as Cmp<B>>::Output;
    assert_eq!(<P5Cmp_0 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P6 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP1 = <<A as Add<B>>::Output as Same<P6>>::Output;

    assert_eq!(<P5AddP1 as Integer>::to_i64(), <P6 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP1 = <<A as Sub<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5SubP1 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP1 = <<A as Mul<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MulP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP1 = <<A as Min<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5MinP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Max_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP1 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP1 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP1 = <<A as Div<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5DivP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP1 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP1 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
```

```
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP1 = <<A as PartialDiv<B>>::Output as Same<P5>>::Outpu

    assert_eq!(<P5PartialDivP1 as Integer>::to_i64(), <P5 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5PowP1 = <<A as Pow<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5PowP1 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P1() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP1 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP1 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P7 = PInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP2 = <<A as Add<B>>::Output as Same<P7>>::Output;

    assert_eq!(<P5AddP2 as Integer>::to_i64(), <P7 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP2 = <<A as Sub<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5SubP2 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P10 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MulP2 = <<A as Mul<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5MulP2 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP2 = <<A as Min<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5MinP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP2 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP2 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP2 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P2() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5DivP2 = <<A as Div<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5DivP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP2 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP2 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5PowP2 = <<A as Pow<B>>::Output as Same<P25>>::Output;

    assert_eq!(<P5PowP2 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P2() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP2 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP2 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P8 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP3 = <<A as Add<B>>::Output as Same<P8>>::Output;
```

```
    assert_eq!(<P5AddP3 as Integer>::to_i64(), <P8 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5SubP3 = <<A as Sub<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5SubP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P15 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MulP3 = <<A as Mul<B>>::Output as Same<P15>>::Output;

    assert_eq!(<P5MulP3 as Integer>::to_i64(), <P15 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP3 = <<A as Min<B>>::Output as Same<P3>>::Output;

    assert_eq!(<P5MinP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP3 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP3 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Gcd_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP3 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP3 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP3 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5RemP3 = <<A as Rem<B>>::Output as Same<P2>>::Output;

    assert_eq!(<P5RemP3 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B1

    #[allow(non_camel_case_types)]
    type P5PowP3 = <<A as Pow<B>>::Output as Same<P125>>::Output;

    assert_eq!(<P5PowP3 as Integer>::to_i64(), <P125 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P3() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UTerm, B1>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type P5CmpP3 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP3 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P9 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5AddP4 = <<A as Add<B>>::Output as Same<P9>>::Output;

    assert_eq!(<P5AddP4 as Integer>::to_i64(), <P9 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5SubP4 = <<A as Sub<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5SubP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P20 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>, B0>

    #[allow(non_camel_case_types)]
    type P5MulP4 = <<A as Mul<B>>::Output as Same<P20>>::Output;

    assert_eq!(<P5MulP4 as Integer>::to_i64(), <P20 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5MinP4 = <<A as Min<B>>::Output as Same<P4>>::Output;

    assert_eq!(<P5MinP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP4 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP4 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP4 = <<A as Gcd<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5GcdP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP4 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Rem_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5RemP4 = <<A as Rem<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5RemP4 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P4() {
```

```
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P625 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UTer

    #[allow(non_camel_case_types)]
    type P5PowP4 = <<A as Pow<B>>::Output as Same<P625>>::Output;

    assert_eq!(<P5PowP4 as Integer>::to_i64(), <P625 as Integer>::to_i64())
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P4() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type P5CmpP4 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP4 as Ord>::to_ordering(), Ordering::Greater);
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Add_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P10 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type P5AddP5 = <<A as Add<B>>::Output as Same<P10>>::Output;

    assert_eq!(<P5AddP5 as Integer>::to_i64(), <P10 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Sub_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5SubP5 = <<A as Sub<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5SubP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Mul_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P25 = PInt<UInt<UInt<UInt<UInt<UTerm, B1>, B1>, B0>, B0>, B1>

    #[allow(non_camel_case_types)]
    type P5MulP5 = <<A as Mul<B>>::Output as Same<P25>>::Output;
```

```
    assert_eq!(<P5MulP5 as Integer>::to_i64(), <P25 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Min_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MinP5 = <<A as Min<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MinP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Max_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5MaxP5 = <<A as Max<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5MaxP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Gcd_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5GcdP5 = <<A as Gcd<B>>::Output as Same<P5>>::Output;

    assert_eq!(<P5GcdP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Div_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5DivP5 = <<A as Div<B>>::Output as Same<P1>>::Output;

    assert_eq!(<P5DivP5 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test_P5_Rem_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type P5RemP5 = <<A as Rem<B>>::Output as Same<_0>>::Output;

    assert_eq!(<P5RemP5 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_PartialDiv_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type P5PartialDivP5 = <<A as PartialDiv<B>>::Output as Same<P1>>::Outpu

    assert_eq!(<P5PartialDivP5 as Integer>::to_i64(), <P1 as Integer>::to_i
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Pow_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P3125 = PInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UInt<UIn

    #[allow(non_camel_case_types)]
    type P5PowP5 = <<A as Pow<B>>::Output as Same<P3125>>::Output;

    assert_eq!(<P5PowP5 as Integer>::to_i64(), <P3125 as Integer>::to_i64()
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Cmp_P5() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type B = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type P5CmpP5 = <A as Cmp<B>>::Output;
    assert_eq!(<P5CmpP5 as Ord>::to_ordering(), Ordering::Equal);
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
```

```
    type NegN5 = <<A as Neg>::Output as Same<P5>>::Output;
    assert_eq!(<NegN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N5_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsN5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Neg() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN4 = <<A as Neg>::Output as Same<P4>>::Output;
    assert_eq!(<NegN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N4_Abs() {
    type A = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsN4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegN3 = <<A as Neg>::Output as Same<P3>>::Output;
    assert_eq!(<NegN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N3_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsN3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
```

```
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Neg() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegN2 = <<A as Neg>::Output as Same<P2>>::Output;
    assert_eq!(<NegN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N2_Abs() {
    type A = NInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsN2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsN2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Neg() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegN1 = <<A as Neg>::Output as Same<P1>>::Output;
    assert_eq!(<NegN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_N1_Abs() {
    type A = NInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsN1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsN1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test__0_Neg() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Neg_0 = <<A as Neg>::Output as Same<_0>>::Output;
    assert_eq!(<Neg_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
```

```
#[allow(non_snake_case)]
fn test__0_Abs() {
    type A = Z0;
    type _0 = Z0;

    #[allow(non_camel_case_types)]
    type Abs_0 = <<A as Abs>::Output as Same<_0>>::Output;
    assert_eq!(<Abs_0 as Integer>::to_i64(), <_0 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Neg() {
    type A = PInt<UInt<UTerm, B1>>;
    type N1 = NInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type NegP1 = <<A as Neg>::Output as Same<N1>>::Output;
    assert_eq!(<NegP1 as Integer>::to_i64(), <N1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P1_Abs() {
    type A = PInt<UInt<UTerm, B1>>;
    type P1 = PInt<UInt<UTerm, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP1 = <<A as Abs>::Output as Same<P1>>::Output;
    assert_eq!(<AbsP1 as Integer>::to_i64(), <P1 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Neg() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type N2 = NInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP2 = <<A as Neg>::Output as Same<N2>>::Output;
    assert_eq!(<NegP2 as Integer>::to_i64(), <N2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P2_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B0>>;
    type P2 = PInt<UInt<UInt<UTerm, B1>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP2 = <<A as Abs>::Output as Same<P2>>::Output;
    assert_eq!(<AbsP2 as Integer>::to_i64(), <P2 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Neg() {
```

```
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type N3 = NInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP3 = <<A as Neg>::Output as Same<N3>>::Output;
    assert_eq!(<NegP3 as Integer>::to_i64(), <N3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P3_Abs() {
    type A = PInt<UInt<UInt<UTerm, B1>, B1>>;
    type P3 = PInt<UInt<UInt<UTerm, B1>, B1>>;

    #[allow(non_camel_case_types)]
    type AbsP3 = <<A as Abs>::Output as Same<P3>>::Output;
    assert_eq!(<AbsP3 as Integer>::to_i64(), <P3 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type N4 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type NegP4 = <<A as Neg>::Output as Same<N4>>::Output;
    assert_eq!(<NegP4 as Integer>::to_i64(), <N4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P4_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;
    type P4 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B0>>;

    #[allow(non_camel_case_types)]
    type AbsP4 = <<A as Abs>::Output as Same<P4>>::Output;
    assert_eq!(<AbsP4 as Integer>::to_i64(), <P4 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Neg() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type N5 = NInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;

    #[allow(non_camel_case_types)]
    type NegP5 = <<A as Neg>::Output as Same<N5>>::Output;
    assert_eq!(<NegP5 as Integer>::to_i64(), <N5 as Integer>::to_i64());
}
#[test]
#[allow(non_snake_case)]
fn test_P5_Abs() {
    type A = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
    type P5 = PInt<UInt<UInt<UInt<UTerm, B1>, B0>, B1>>;
```

```
    #[allow(non_camel_case_types)]
    type AbsP5 = <<A as Abs>::Output as Same<P5>>::Output;
    assert_eq!(<AbsP5 as Integer>::to_i64(), <P5 as Integer>::to_i64());
}
```

File: ./target/debug/build/typenum-0568521468b798f9/out/op.rs

```
/**
Convenient type operations.

Any types representing values must be able to be expressed as `ident`s. Tha
in scope.

For example, `P5` is okay, but `typenum::P5` is not.

You may combine operators arbitrarily, although doing so excessively may re
recursion limit.

# Example
```rust
#![recursion_limit="128"]
#[macro_use] extern crate typenum;
use typenum::consts::*;

fn main() {
    assert_type!(
        op!(min((P1 - P2) * (N3 + N7), P5 * (P3 + P4)) == P10)
    );
}
```
Operators are evaluated based on the operator precedence outlined
[here](https://doc.rust-lang.org/reference.html#operator-precedence).

The full list of supported operators and functions is as follows:

`*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `&`, `^`, `|`, `==`, `!=`, `<=`, `>=`,

They all expand to type aliases defined in the `operator_aliases` module. H
including examples:

Operator `*`. Expands to `Prod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 * P3), P6);
```
```

```
# }
```

Operator `/`. Expands to `Quot`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 / P2), P3);
# }
```

Operator `%`. Expands to `Mod`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 % P3), P2);
# }
```

Operator `+`. Expands to `Sum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 + P3), P5);
# }
```

Operator `-`. Expands to `Diff`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P2 - P3), N1);
# }
```

Operator `<<`. Expands to `Shleft`.

```rust
# #[macro_use] extern crate typenum;
```

```
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U1 << U5), U32);
# }
```

Operator `>>`. Expands to `Shright`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U32 >> U5), U1);
# }
```

Operator `&`. Expands to `And`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 & U3), U1);
# }
```

Operator `^`. Expands to `Xor`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 ^ U3), U6);
# }
```

Operator `|`. Expands to `Or`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(U5 | U3), U7);
# }
```

Operator `==`. Expands to `Eq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 == P3 + P2), True);
# }
```

Operator `!=`. Expands to `NotEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 != P3 + P2), False);
# }
```

Operator `<=`. Expands to `LeEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 <= P3 + P2), False);
# }
```

Operator `>=`. Expands to `GrEq`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P6 >= P3 + P2), True);
# }
```

Operator `<`. Expands to `Le`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P4 < P3 + P2), True);
# }
```

Operator `>`. Expands to `Gr`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(P5 < P3 + P2), False);
# }
```

Operator `cmp`. Expands to `Compare`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cmp(P2, P3)), Less);
# }
```

Operator `sqr`. Expands to `Square`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqr(P2)), P4);
# }
```

Operator `sqrt`. Expands to `Sqrt`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(sqrt(U9)), U3);
# }
```

Operator `abs`. Expands to `AbsVal`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
```

```
assert_type_eq!(op!(abs(N2)), P2);
# }
```

Operator `cube`. Expands to `Cube`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(cube(P2)), P8);
# }
```

Operator `pow`. Expands to `Exp`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(pow(P2, P3)), P8);
# }
```

Operator `min`. Expands to `Minimum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(min(P2, P3)), P2);
# }
```

Operator `max`. Expands to `Maximum`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(max(P2, P3)), P3);
# }
```

Operator `log2`. Expands to `Log2`.

```rust
```

```
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(log2(U9)), U3);
# }
```

Operator `gcd`. Expands to `Gcf`.

```rust
# #[macro_use] extern crate typenum;
# use typenum::*;
# fn main() {
assert_type_eq!(op!(gcd(U9, U21)), U3);
# }
```

*/
#[macro_export(local_inner_macros)]
macro_rules! op {
    ($($tail:tt)*) => (__op_internal__!($($tail)*));
}

    #[doc(hidden)]
    #[macro_export(local_inner_macros)]
    macro_rules! __op_internal__ {

(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cmp $($tail:tt)
    __op_internal__!(@stack[Compare, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqr $($tail:tt)
    __op_internal__!(@stack[Square, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: sqrt $($tail:tt
    __op_internal__!(@stack[Sqrt, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: abs $($tail:tt)
    __op_internal__!(@stack[AbsVal, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: cube $($tail:tt
    __op_internal__!(@stack[Cube, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: pow $($tail:tt)
    __op_internal__!(@stack[Exp, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: min $($tail:tt)
    __op_internal__!(@stack[Minimum, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: max $($tail:tt)
    __op_internal__!(@stack[Maximum, $($stack,)*] @queue[$($queue,)*] @tail
);

```
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: log2 $($tail:tt
    __op_internal__!(@stack[Log2, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: gcd $($tail:tt)
    __op_internal__!(@stack[Gcf, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: , $($ta
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: *
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: *
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: *
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: * $($tail:tt)*)
    __op_internal__!(@stack[Prod, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: /
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: /
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: /
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: / $($tail:tt)*)
    __op_internal__!(@stack[Quot, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: %
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: %
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: %
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: % $($tail:tt)*)
    __op_internal__!(@stack[Mod, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: +
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: +
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: +
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: +
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: +
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: + $($tail:tt)*)
    __op_internal__!(@stack[Sum, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: -
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: -
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: -
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: -
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: -
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: - $($tail:tt)*)
    __op_internal__!(@stack[Diff, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <<
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <<
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
```

```
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: << $($tail:tt)*
    __op_internal__!(@stack[Shleft, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >>
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >>
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >> $($tail:tt)*
    __op_internal__!(@stack[Shright, $($stack,)*] @queue[$($queue,)*] @tail
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: &
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: &
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: &
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: &
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: &
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: &
);
```

```
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: & $($tail:tt)*)
    __op_internal__!(@stack[And, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: ^
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: ^
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ^
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ^
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: ^
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ^
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ^
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ^ $($tail:tt)*)
    __op_internal__!(@stack[Xor, $($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: |
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: |
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: |
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: |
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: |
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($t
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: |
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: |
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: | $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: | $($tail:tt)*)
    __op_internal__!(@stack[Or, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: =
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: =
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: ==
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: ==
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: =
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: ==
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: ==
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: ==
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: ==
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: =
```

```
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: =
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: ==
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: ==
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: == $($tail:tt)*
    __op_internal__!(@stack[Eq, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: !
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: !
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: !=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: !=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: !
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: !=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: !=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: !=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: !=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: !
);
```

```
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: !
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: !=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: !=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: != $($tail:tt)*
    __op_internal__!(@stack[NotEq, $($stack,)*] @queue[$($queue,)*] @tail:
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: <=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: <=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tai
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: <=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: <=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: <= $($tail:tt)*
    __op_internal__!(@stack[LeEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >=
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >=
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >=
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >=
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: >=
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: >=
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
```

```
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: >=
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: >=
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: >= $($tail:tt)*
    __op_internal__!(@stack[GrEq, $($stack,)*] @queue[$($queue,)*] @tail: $
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: <
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: <
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: <
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: <
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: <
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: <
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: <
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: < $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: < $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: <
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: <
);
```

```
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: < $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: < $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: < $($tail:tt)*)
    __op_internal__!(@stack[Le, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[Prod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Prod, $($queue,)*] @tail: >
);
(@stack[Quot, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Quot, $($queue,)*] @tail: >
);
(@stack[Mod, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Mod, $($queue,)*] @tail: >
);
(@stack[Sum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Sum, $($queue,)*] @tail: >
);
(@stack[Diff, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[Diff, $($queue,)*] @tail: >
);
(@stack[Shleft, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Shleft, $($queue,)*] @tail:
);
(@stack[Shright, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Shright, $($queue,)*] @tail
);
(@stack[And, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[And, $($queue,)*] @tail: >
);
(@stack[Xor, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:
    __op_internal__!(@stack[$($stack,)*] @queue[Xor, $($queue,)*] @tail: >
);
(@stack[Or, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Or, $($queue,)*] @tail: > $
);
(@stack[Eq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Eq, $($queue,)*] @tail: > $
);
(@stack[NotEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tai
    __op_internal__!(@stack[$($stack,)*] @queue[NotEq, $($queue,)*] @tail:
);
(@stack[LeEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[LeEq, $($queue,)*] @tail: >
);
(@stack[GrEq, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail
    __op_internal__!(@stack[$($stack,)*] @queue[GrEq, $($queue,)*] @tail: >
);
(@stack[Le, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
```

```
    __op_internal__!(@stack[$($stack,)*] @queue[Le, $($queue,)*] @tail: > $
);
(@stack[Gr, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:t
    __op_internal__!(@stack[$($stack,)*] @queue[Gr, $($queue,)*] @tail: > $
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: > $($tail:tt)*)
    __op_internal__!(@stack[Gr, $($stack,)*] @queue[$($queue,)*] @tail: $($
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: ( $($stuff:tt)*
 => (
    __op_internal__!(@stack[LParen, $($stack,)*] @queue[$($queue,)*]
                     @tail: $($stuff)* RParen $($tail)*)
);
(@stack[LParen, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: RParen
    __op_internal__!(@rp3 @stack[$($stack,)*] @queue[$($queue,)*] @tail: $(
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
 => (
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@rp3 @stack[Compare, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Compare, $($queue,)*] @tail
);
(@rp3 @stack[Square, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[Square, $($queue,)*] @tail:
);
(@rp3 @stack[Sqrt, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Sqrt, $($queue,)*] @tail: $
);
(@rp3 @stack[AbsVal, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $(
    __op_internal__!(@stack[$($stack,)*] @queue[AbsVal, $($queue,)*] @tail:
);
(@rp3 @stack[Cube, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Cube, $($queue,)*] @tail: $
);
(@rp3 @stack[Exp, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Exp, $($queue,)*] @tail: $(
);
(@rp3 @stack[Minimum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Minimum, $($queue,)*] @tail
);
(@rp3 @stack[Maximum, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $
    __op_internal__!(@stack[$($stack,)*] @queue[Maximum, $($queue,)*] @tail
);
(@rp3 @stack[Log2, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($t
    __op_internal__!(@stack[$($stack,)*] @queue[Log2, $($queue,)*] @tail: $
);
(@rp3 @stack[Gcf, $($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($ta
    __op_internal__!(@stack[$($stack,)*] @queue[Gcf, $($queue,)*] @tail: $(
);
(@rp3 @stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $($tail:tt
    __op_internal__!(@stack[$($stack,)*] @queue[$($queue,)*] @tail: $($tail
```

```
);
(@stack[$($stack:ident,)*] @queue[$($queue:ident,)*] @tail: $num:ident $($t
    __op_internal__!(@stack[$($stack,)*] @queue[$num, $($queue,)*] @tail: $
);
(@stack[] @queue[$($queue:ident,)*] @tail: ) => (
    __op_internal__!(@reverse[] @input: $($queue,)*)
);
(@stack[$stack_top:ident, $($stack:ident,)*] @queue[$($queue:ident,)*] @tai
    __op_internal__!(@stack[$($stack,)*] @queue[$stack_top, $($queue,)*] @t
);
(@reverse[$($revved:ident,)*] @input: $head:ident, $($tail:ident,)* ) => (
    __op_internal__!(@reverse[$head, $($revved,)*] @input: $($tail,)*)
);
(@reverse[$($revved:ident,)*] @input: ) => (
    __op_internal__!(@eval @stack[] @input[$($revved,)*])
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Prod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Prod<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Quot, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Quot<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Mod, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Mod<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Sum, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Sum<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Diff, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Diff<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shleft, $($tail:ident,)*
    __op_internal__!(@eval @stack[$crate::Shleft<$b, $a>, $($stack,)*] @inp
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Shright, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Shright<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[And, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::And<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Xor, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Xor<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Or, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Or<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Eq, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Eq<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[NotEq, $($tail:ident,)*]
    __op_internal__!(@eval @stack[$crate::NotEq<$b, $a>, $($stack,)*] @inpu
);
```

```
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[LeEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::LeEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[GrEq, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::GrEq<$b, $a>, $($stack,)*] @input
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Le, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Le<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gr, $($tail:ident,)*]) =
    __op_internal__!(@eval @stack[$crate::Gr<$b, $a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Compare, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Compare<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Exp, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Exp<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Minimum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Minimum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Maximum, $($tail:ident,)
    __op_internal__!(@eval @stack[$crate::Maximum<$b, $a>, $($stack,)*] @in
);
(@eval @stack[$a:ty, $b:ty, $($stack:ty,)*] @input[Gcf, $($tail:ident,)*])
    __op_internal__!(@eval @stack[$crate::Gcf<$b, $a>, $($stack,)*] @input[
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Square, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Square<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Sqrt, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Sqrt<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[AbsVal, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::AbsVal<$a>, $($stack,)*] @input[$
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Cube, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Cube<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$a:ty, $($stack:ty,)*] @input[Log2, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$crate::Log2<$a>, $($stack,)*] @input[$($
);
(@eval @stack[$($stack:ty,)*] @input[$head:ident, $($tail:ident,)*]) => (
    __op_internal__!(@eval @stack[$head, $($stack,)*] @input[$($tail,)*])
);
(@eval @stack[$stack:ty,] @input[]) => (
    $stack
);
($($tail:tt)* ) => (
    __op_internal__!(@stack[] @queue[] @tail: $($tail)*)
);
}
```

File: ./target/debug/build/libsqlite3-sys-bf468e800d19c02b/out/bindg

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
```

```
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
```

```
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
```

```
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
```

```
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
```

```
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
```

```
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
```

```
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
```

```
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
```

```
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
```

```
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xWrite: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *const ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
            iOfst: sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
```

```
pub xTruncate: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xDeviceCharacteristics: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
>,
pub xShmMap: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        iPg: ::std::os::raw::c_int,
        pgsz: ::std::os::raw::c_int,
        arg2: ::std::os::raw::c_int,
```

```
            arg3: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmLock: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            offset: ::std::os::raw::c_int,
            n: ::std::os::raw::c_int,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
```

```
pub pNext: *mut sqlite3_vfs,
pub zName: *const ::std::os::raw::c_char,
pub pAppData: *mut ::std::os::raw::c_void,
pub xOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: sqlite3_filename,
        arg2: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
        pOutFlags: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xDelete: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        syncDir: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xAccess: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        flags: ::std::os::raw::c_int,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFullPathname: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        nOut: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xDlOpen: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zFilename: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
```

```
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
>,
pub xRandomness: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSleep: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        microseconds: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTime: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
>,
pub xGetLastError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTimeInt64: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xSetSystemCall: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
        arg2: sqlite3_syscall_ptr,
    ) -> ::std::os::raw::c_int,
>,
pub xGetSystemCall: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        zName: *const ::std::os::raw::c_char,
```

```
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
```

```
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
```

```
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
```

```
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
```

```
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
```

```
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
```

```
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
```

```
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
```

```
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
```

```
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
```

```
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
```

```
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
```

```
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
```

```
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
```

```
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
```

```
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
```

```
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
```

```
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
```

```
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
```

```
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
```

```
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xConnect: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xBestIndex: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: *mut sqlite3_index_info,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDisconnect: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xDestroy: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            ppCursor: *mut *mut sqlite3_vtab_cursor,
        ) -> ::std::os::raw::c_int,
    >,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
    >,
    pub xFilter: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vtab_cursor,
            idxNum: ::std::os::raw::c_int,
            idxStr: *const ::std::os::raw::c_char,
            argc: ::std::os::raw::c_int,
            argv: *mut *mut sqlite3_value,
        ) -> ::std::os::raw::c_int,
    >,
```

```
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
        pVtab: *mut sqlite3_vtab,
        nArg: ::std::os::raw::c_int,
        zName: *const ::std::os::raw::c_char,
        pxFunc: *mut ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        ppArg: *mut *mut ::std::os::raw::c_void,
```

```
        ) -> ::std::os::raw::c_int,
    >,
    pub xRename: ::std::option::Option<
        unsafe extern "C" fn(
            pVtab: *mut sqlite3_vtab,
            zNew: *const ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSavepoint: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
```

```
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
```

```
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
```

```
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
```

```
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
```

```
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
```

```
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
```

```
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
```

```
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
```

```
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
```

```
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
```

```
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
```

```
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
```

```
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
```

```
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
```

```
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
```

```
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
```

```
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
```

```
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
    pub xUserData: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
    >,
    pub xColumnCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xRowCount: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pnRow: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xColumnTotalSize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iCol: ::std::os::raw::c_int,
            pnToken: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            pCtx: *mut ::std::os::raw::c_void,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    arg1: *mut ::std::os::raw::c_void,
                    arg2: ::std::os::raw::c_int,
                    arg3: *const ::std::os::raw::c_char,
                    arg4: ::std::os::raw::c_int,
                    arg5: ::std::os::raw::c_int,
                    arg6: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xPhraseSize: ::std::option::Option<
        unsafe extern "C" fn(
```

```
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            arg1: *mut Fts5Context,
            pAux: *mut ::std::os::raw::c_void,
            xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetAuxdata: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            bClear: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xPhraseFirst: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
            arg4: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNext: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
            piOff: *mut ::std::os::raw::c_int,
        ),
    >,
    pub xPhraseFirstColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iPhrase: ::std::os::raw::c_int,
            arg2: *mut Fts5PhraseIter,
            arg3: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xPhraseNextColumn: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            arg2: *mut Fts5PhraseIter,
            piCol: *mut ::std::os::raw::c_int,
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
```

```
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
```

```
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./target/debug/build/libsqlite3-sys-3c1ce36a9e5afc9c/out/bindg

```
/* automatically generated by rust-bindgen 0.64.0 */

pub const SQLITE_VERSION: &[u8; 7usize] = b"3.41.2\0";
pub const SQLITE_VERSION_NUMBER: i32 = 3041002;
pub const SQLITE_SOURCE_ID: &[u8; 85usize] =
    b"2023-03-22 11:56:21 0d1fc92f94cb6b76bffe3ec34d69cffde2924203304e8ffc4
pub const SQLITE_OK: i32 = 0;
pub const SQLITE_ERROR: i32 = 1;
pub const SQLITE_INTERNAL: i32 = 2;
pub const SQLITE_PERM: i32 = 3;
pub const SQLITE_ABORT: i32 = 4;
pub const SQLITE_BUSY: i32 = 5;
pub const SQLITE_LOCKED: i32 = 6;
pub const SQLITE_NOMEM: i32 = 7;
pub const SQLITE_READONLY: i32 = 8;
pub const SQLITE_INTERRUPT: i32 = 9;
pub const SQLITE_IOERR: i32 = 10;
pub const SQLITE_CORRUPT: i32 = 11;
pub const SQLITE_NOTFOUND: i32 = 12;
pub const SQLITE_FULL: i32 = 13;
pub const SQLITE_CANTOPEN: i32 = 14;
pub const SQLITE_PROTOCOL: i32 = 15;
pub const SQLITE_EMPTY: i32 = 16;
pub const SQLITE_SCHEMA: i32 = 17;
pub const SQLITE_TOOBIG: i32 = 18;
pub const SQLITE_CONSTRAINT: i32 = 19;
pub const SQLITE_MISMATCH: i32 = 20;
pub const SQLITE_MISUSE: i32 = 21;
pub const SQLITE_NOLFS: i32 = 22;
pub const SQLITE_AUTH: i32 = 23;
pub const SQLITE_FORMAT: i32 = 24;
pub const SQLITE_RANGE: i32 = 25;
pub const SQLITE_NOTADB: i32 = 26;
pub const SQLITE_NOTICE: i32 = 27;
pub const SQLITE_WARNING: i32 = 28;
pub const SQLITE_ROW: i32 = 100;
pub const SQLITE_DONE: i32 = 101;
pub const SQLITE_ERROR_MISSING_COLLSEQ: i32 = 257;
pub const SQLITE_ERROR_RETRY: i32 = 513;
```

```
pub const SQLITE_ERROR_SNAPSHOT: i32 = 769;
pub const SQLITE_IOERR_READ: i32 = 266;
pub const SQLITE_IOERR_SHORT_READ: i32 = 522;
pub const SQLITE_IOERR_WRITE: i32 = 778;
pub const SQLITE_IOERR_FSYNC: i32 = 1034;
pub const SQLITE_IOERR_DIR_FSYNC: i32 = 1290;
pub const SQLITE_IOERR_TRUNCATE: i32 = 1546;
pub const SQLITE_IOERR_FSTAT: i32 = 1802;
pub const SQLITE_IOERR_UNLOCK: i32 = 2058;
pub const SQLITE_IOERR_RDLOCK: i32 = 2314;
pub const SQLITE_IOERR_DELETE: i32 = 2570;
pub const SQLITE_IOERR_BLOCKED: i32 = 2826;
pub const SQLITE_IOERR_NOMEM: i32 = 3082;
pub const SQLITE_IOERR_ACCESS: i32 = 3338;
pub const SQLITE_IOERR_CHECKRESERVEDLOCK: i32 = 3594;
pub const SQLITE_IOERR_LOCK: i32 = 3850;
pub const SQLITE_IOERR_CLOSE: i32 = 4106;
pub const SQLITE_IOERR_DIR_CLOSE: i32 = 4362;
pub const SQLITE_IOERR_SHMOPEN: i32 = 4618;
pub const SQLITE_IOERR_SHMSIZE: i32 = 4874;
pub const SQLITE_IOERR_SHMLOCK: i32 = 5130;
pub const SQLITE_IOERR_SHMMAP: i32 = 5386;
pub const SQLITE_IOERR_SEEK: i32 = 5642;
pub const SQLITE_IOERR_DELETE_NOENT: i32 = 5898;
pub const SQLITE_IOERR_MMAP: i32 = 6154;
pub const SQLITE_IOERR_GETTEMPPATH: i32 = 6410;
pub const SQLITE_IOERR_CONVPATH: i32 = 6666;
pub const SQLITE_IOERR_VNODE: i32 = 6922;
pub const SQLITE_IOERR_AUTH: i32 = 7178;
pub const SQLITE_IOERR_BEGIN_ATOMIC: i32 = 7434;
pub const SQLITE_IOERR_COMMIT_ATOMIC: i32 = 7690;
pub const SQLITE_IOERR_ROLLBACK_ATOMIC: i32 = 7946;
pub const SQLITE_IOERR_DATA: i32 = 8202;
pub const SQLITE_IOERR_CORRUPTFS: i32 = 8458;
pub const SQLITE_LOCKED_SHAREDCACHE: i32 = 262;
pub const SQLITE_LOCKED_VTAB: i32 = 518;
pub const SQLITE_BUSY_RECOVERY: i32 = 261;
pub const SQLITE_BUSY_SNAPSHOT: i32 = 517;
pub const SQLITE_BUSY_TIMEOUT: i32 = 773;
pub const SQLITE_CANTOPEN_NOTEMPDIR: i32 = 270;
pub const SQLITE_CANTOPEN_ISDIR: i32 = 526;
pub const SQLITE_CANTOPEN_FULLPATH: i32 = 782;
pub const SQLITE_CANTOPEN_CONVPATH: i32 = 1038;
pub const SQLITE_CANTOPEN_DIRTYWAL: i32 = 1294;
pub const SQLITE_CANTOPEN_SYMLINK: i32 = 1550;
pub const SQLITE_CORRUPT_VTAB: i32 = 267;
pub const SQLITE_CORRUPT_SEQUENCE: i32 = 523;
pub const SQLITE_CORRUPT_INDEX: i32 = 779;
pub const SQLITE_READONLY_RECOVERY: i32 = 264;
pub const SQLITE_READONLY_CANTLOCK: i32 = 520;
pub const SQLITE_READONLY_ROLLBACK: i32 = 776;
pub const SQLITE_READONLY_DBMOVED: i32 = 1032;
```

```
pub const SQLITE_READONLY_CANTINIT: i32 = 1288;
pub const SQLITE_READONLY_DIRECTORY: i32 = 1544;
pub const SQLITE_ABORT_ROLLBACK: i32 = 516;
pub const SQLITE_CONSTRAINT_CHECK: i32 = 275;
pub const SQLITE_CONSTRAINT_COMMITHOOK: i32 = 531;
pub const SQLITE_CONSTRAINT_FOREIGNKEY: i32 = 787;
pub const SQLITE_CONSTRAINT_FUNCTION: i32 = 1043;
pub const SQLITE_CONSTRAINT_NOTNULL: i32 = 1299;
pub const SQLITE_CONSTRAINT_PRIMARYKEY: i32 = 1555;
pub const SQLITE_CONSTRAINT_TRIGGER: i32 = 1811;
pub const SQLITE_CONSTRAINT_UNIQUE: i32 = 2067;
pub const SQLITE_CONSTRAINT_VTAB: i32 = 2323;
pub const SQLITE_CONSTRAINT_ROWID: i32 = 2579;
pub const SQLITE_CONSTRAINT_PINNED: i32 = 2835;
pub const SQLITE_CONSTRAINT_DATATYPE: i32 = 3091;
pub const SQLITE_NOTICE_RECOVER_WAL: i32 = 283;
pub const SQLITE_NOTICE_RECOVER_ROLLBACK: i32 = 539;
pub const SQLITE_NOTICE_RBU: i32 = 795;
pub const SQLITE_WARNING_AUTOINDEX: i32 = 284;
pub const SQLITE_AUTH_USER: i32 = 279;
pub const SQLITE_OK_LOAD_PERMANENTLY: i32 = 256;
pub const SQLITE_OK_SYMLINK: i32 = 512;
pub const SQLITE_OPEN_READONLY: i32 = 1;
pub const SQLITE_OPEN_READWRITE: i32 = 2;
pub const SQLITE_OPEN_CREATE: i32 = 4;
pub const SQLITE_OPEN_DELETEONCLOSE: i32 = 8;
pub const SQLITE_OPEN_EXCLUSIVE: i32 = 16;
pub const SQLITE_OPEN_AUTOPROXY: i32 = 32;
pub const SQLITE_OPEN_URI: i32 = 64;
pub const SQLITE_OPEN_MEMORY: i32 = 128;
pub const SQLITE_OPEN_MAIN_DB: i32 = 256;
pub const SQLITE_OPEN_TEMP_DB: i32 = 512;
pub const SQLITE_OPEN_TRANSIENT_DB: i32 = 1024;
pub const SQLITE_OPEN_MAIN_JOURNAL: i32 = 2048;
pub const SQLITE_OPEN_TEMP_JOURNAL: i32 = 4096;
pub const SQLITE_OPEN_SUBJOURNAL: i32 = 8192;
pub const SQLITE_OPEN_SUPER_JOURNAL: i32 = 16384;
pub const SQLITE_OPEN_NOMUTEX: i32 = 32768;
pub const SQLITE_OPEN_FULLMUTEX: i32 = 65536;
pub const SQLITE_OPEN_SHAREDCACHE: i32 = 131072;
pub const SQLITE_OPEN_PRIVATECACHE: i32 = 262144;
pub const SQLITE_OPEN_WAL: i32 = 524288;
pub const SQLITE_OPEN_NOFOLLOW: i32 = 16777216;
pub const SQLITE_OPEN_EXRESCODE: i32 = 33554432;
pub const SQLITE_OPEN_MASTER_JOURNAL: i32 = 16384;
pub const SQLITE_IOCAP_ATOMIC: i32 = 1;
pub const SQLITE_IOCAP_ATOMIC512: i32 = 2;
pub const SQLITE_IOCAP_ATOMIC1K: i32 = 4;
pub const SQLITE_IOCAP_ATOMIC2K: i32 = 8;
pub const SQLITE_IOCAP_ATOMIC4K: i32 = 16;
pub const SQLITE_IOCAP_ATOMIC8K: i32 = 32;
pub const SQLITE_IOCAP_ATOMIC16K: i32 = 64;
```

```
pub const SQLITE_IOCAP_ATOMIC32K: i32 = 128;
pub const SQLITE_IOCAP_ATOMIC64K: i32 = 256;
pub const SQLITE_IOCAP_SAFE_APPEND: i32 = 512;
pub const SQLITE_IOCAP_SEQUENTIAL: i32 = 1024;
pub const SQLITE_IOCAP_UNDELETABLE_WHEN_OPEN: i32 = 2048;
pub const SQLITE_IOCAP_POWERSAFE_OVERWRITE: i32 = 4096;
pub const SQLITE_IOCAP_IMMUTABLE: i32 = 8192;
pub const SQLITE_IOCAP_BATCH_ATOMIC: i32 = 16384;
pub const SQLITE_LOCK_NONE: i32 = 0;
pub const SQLITE_LOCK_SHARED: i32 = 1;
pub const SQLITE_LOCK_RESERVED: i32 = 2;
pub const SQLITE_LOCK_PENDING: i32 = 3;
pub const SQLITE_LOCK_EXCLUSIVE: i32 = 4;
pub const SQLITE_SYNC_NORMAL: i32 = 2;
pub const SQLITE_SYNC_FULL: i32 = 3;
pub const SQLITE_SYNC_DATAONLY: i32 = 16;
pub const SQLITE_FCNTL_LOCKSTATE: i32 = 1;
pub const SQLITE_FCNTL_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_FCNTL_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_FCNTL_LAST_ERRNO: i32 = 4;
pub const SQLITE_FCNTL_SIZE_HINT: i32 = 5;
pub const SQLITE_FCNTL_CHUNK_SIZE: i32 = 6;
pub const SQLITE_FCNTL_FILE_POINTER: i32 = 7;
pub const SQLITE_FCNTL_SYNC_OMITTED: i32 = 8;
pub const SQLITE_FCNTL_WIN32_AV_RETRY: i32 = 9;
pub const SQLITE_FCNTL_PERSIST_WAL: i32 = 10;
pub const SQLITE_FCNTL_OVERWRITE: i32 = 11;
pub const SQLITE_FCNTL_VFSNAME: i32 = 12;
pub const SQLITE_FCNTL_POWERSAFE_OVERWRITE: i32 = 13;
pub const SQLITE_FCNTL_PRAGMA: i32 = 14;
pub const SQLITE_FCNTL_BUSYHANDLER: i32 = 15;
pub const SQLITE_FCNTL_TEMPFILENAME: i32 = 16;
pub const SQLITE_FCNTL_MMAP_SIZE: i32 = 18;
pub const SQLITE_FCNTL_TRACE: i32 = 19;
pub const SQLITE_FCNTL_HAS_MOVED: i32 = 20;
pub const SQLITE_FCNTL_SYNC: i32 = 21;
pub const SQLITE_FCNTL_COMMIT_PHASETWO: i32 = 22;
pub const SQLITE_FCNTL_WIN32_SET_HANDLE: i32 = 23;
pub const SQLITE_FCNTL_WAL_BLOCK: i32 = 24;
pub const SQLITE_FCNTL_ZIPVFS: i32 = 25;
pub const SQLITE_FCNTL_RBU: i32 = 26;
pub const SQLITE_FCNTL_VFS_POINTER: i32 = 27;
pub const SQLITE_FCNTL_JOURNAL_POINTER: i32 = 28;
pub const SQLITE_FCNTL_WIN32_GET_HANDLE: i32 = 29;
pub const SQLITE_FCNTL_PDB: i32 = 30;
pub const SQLITE_FCNTL_BEGIN_ATOMIC_WRITE: i32 = 31;
pub const SQLITE_FCNTL_COMMIT_ATOMIC_WRITE: i32 = 32;
pub const SQLITE_FCNTL_ROLLBACK_ATOMIC_WRITE: i32 = 33;
pub const SQLITE_FCNTL_LOCK_TIMEOUT: i32 = 34;
pub const SQLITE_FCNTL_DATA_VERSION: i32 = 35;
pub const SQLITE_FCNTL_SIZE_LIMIT: i32 = 36;
pub const SQLITE_FCNTL_CKPT_DONE: i32 = 37;
```

```
pub const SQLITE_FCNTL_RESERVE_BYTES: i32 = 38;
pub const SQLITE_FCNTL_CKPT_START: i32 = 39;
pub const SQLITE_FCNTL_EXTERNAL_READER: i32 = 40;
pub const SQLITE_FCNTL_CKSM_FILE: i32 = 41;
pub const SQLITE_FCNTL_RESET_CACHE: i32 = 42;
pub const SQLITE_GET_LOCKPROXYFILE: i32 = 2;
pub const SQLITE_SET_LOCKPROXYFILE: i32 = 3;
pub const SQLITE_LAST_ERRNO: i32 = 4;
pub const SQLITE_ACCESS_EXISTS: i32 = 0;
pub const SQLITE_ACCESS_READWRITE: i32 = 1;
pub const SQLITE_ACCESS_READ: i32 = 2;
pub const SQLITE_SHM_UNLOCK: i32 = 1;
pub const SQLITE_SHM_LOCK: i32 = 2;
pub const SQLITE_SHM_SHARED: i32 = 4;
pub const SQLITE_SHM_EXCLUSIVE: i32 = 8;
pub const SQLITE_SHM_NLOCK: i32 = 8;
pub const SQLITE_CONFIG_SINGLETHREAD: i32 = 1;
pub const SQLITE_CONFIG_MULTITHREAD: i32 = 2;
pub const SQLITE_CONFIG_SERIALIZED: i32 = 3;
pub const SQLITE_CONFIG_MALLOC: i32 = 4;
pub const SQLITE_CONFIG_GETMALLOC: i32 = 5;
pub const SQLITE_CONFIG_SCRATCH: i32 = 6;
pub const SQLITE_CONFIG_PAGECACHE: i32 = 7;
pub const SQLITE_CONFIG_HEAP: i32 = 8;
pub const SQLITE_CONFIG_MEMSTATUS: i32 = 9;
pub const SQLITE_CONFIG_MUTEX: i32 = 10;
pub const SQLITE_CONFIG_GETMUTEX: i32 = 11;
pub const SQLITE_CONFIG_LOOKASIDE: i32 = 13;
pub const SQLITE_CONFIG_PCACHE: i32 = 14;
pub const SQLITE_CONFIG_GETPCACHE: i32 = 15;
pub const SQLITE_CONFIG_LOG: i32 = 16;
pub const SQLITE_CONFIG_URI: i32 = 17;
pub const SQLITE_CONFIG_PCACHE2: i32 = 18;
pub const SQLITE_CONFIG_GETPCACHE2: i32 = 19;
pub const SQLITE_CONFIG_COVERING_INDEX_SCAN: i32 = 20;
pub const SQLITE_CONFIG_SQLLOG: i32 = 21;
pub const SQLITE_CONFIG_MMAP_SIZE: i32 = 22;
pub const SQLITE_CONFIG_WIN32_HEAPSIZE: i32 = 23;
pub const SQLITE_CONFIG_PCACHE_HDRSZ: i32 = 24;
pub const SQLITE_CONFIG_PMASZ: i32 = 25;
pub const SQLITE_CONFIG_STMTJRNL_SPILL: i32 = 26;
pub const SQLITE_CONFIG_SMALL_MALLOC: i32 = 27;
pub const SQLITE_CONFIG_SORTERREF_SIZE: i32 = 28;
pub const SQLITE_CONFIG_MEMDB_MAXSIZE: i32 = 29;
pub const SQLITE_DBCONFIG_MAINDBNAME: i32 = 1000;
pub const SQLITE_DBCONFIG_LOOKASIDE: i32 = 1001;
pub const SQLITE_DBCONFIG_ENABLE_FKEY: i32 = 1002;
pub const SQLITE_DBCONFIG_ENABLE_TRIGGER: i32 = 1003;
pub const SQLITE_DBCONFIG_ENABLE_FTS3_TOKENIZER: i32 = 1004;
pub const SQLITE_DBCONFIG_ENABLE_LOAD_EXTENSION: i32 = 1005;
pub const SQLITE_DBCONFIG_NO_CKPT_ON_CLOSE: i32 = 1006;
pub const SQLITE_DBCONFIG_ENABLE_QPSG: i32 = 1007;
```

```
pub const SQLITE_DBCONFIG_TRIGGER_EQP: i32 = 1008;
pub const SQLITE_DBCONFIG_RESET_DATABASE: i32 = 1009;
pub const SQLITE_DBCONFIG_DEFENSIVE: i32 = 1010;
pub const SQLITE_DBCONFIG_WRITABLE_SCHEMA: i32 = 1011;
pub const SQLITE_DBCONFIG_LEGACY_ALTER_TABLE: i32 = 1012;
pub const SQLITE_DBCONFIG_DQS_DML: i32 = 1013;
pub const SQLITE_DBCONFIG_DQS_DDL: i32 = 1014;
pub const SQLITE_DBCONFIG_ENABLE_VIEW: i32 = 1015;
pub const SQLITE_DBCONFIG_LEGACY_FILE_FORMAT: i32 = 1016;
pub const SQLITE_DBCONFIG_TRUSTED_SCHEMA: i32 = 1017;
pub const SQLITE_DBCONFIG_MAX: i32 = 1017;
pub const SQLITE_DENY: i32 = 1;
pub const SQLITE_IGNORE: i32 = 2;
pub const SQLITE_CREATE_INDEX: i32 = 1;
pub const SQLITE_CREATE_TABLE: i32 = 2;
pub const SQLITE_CREATE_TEMP_INDEX: i32 = 3;
pub const SQLITE_CREATE_TEMP_TABLE: i32 = 4;
pub const SQLITE_CREATE_TEMP_TRIGGER: i32 = 5;
pub const SQLITE_CREATE_TEMP_VIEW: i32 = 6;
pub const SQLITE_CREATE_TRIGGER: i32 = 7;
pub const SQLITE_CREATE_VIEW: i32 = 8;
pub const SQLITE_DELETE: i32 = 9;
pub const SQLITE_DROP_INDEX: i32 = 10;
pub const SQLITE_DROP_TABLE: i32 = 11;
pub const SQLITE_DROP_TEMP_INDEX: i32 = 12;
pub const SQLITE_DROP_TEMP_TABLE: i32 = 13;
pub const SQLITE_DROP_TEMP_TRIGGER: i32 = 14;
pub const SQLITE_DROP_TEMP_VIEW: i32 = 15;
pub const SQLITE_DROP_TRIGGER: i32 = 16;
pub const SQLITE_DROP_VIEW: i32 = 17;
pub const SQLITE_INSERT: i32 = 18;
pub const SQLITE_PRAGMA: i32 = 19;
pub const SQLITE_READ: i32 = 20;
pub const SQLITE_SELECT: i32 = 21;
pub const SQLITE_TRANSACTION: i32 = 22;
pub const SQLITE_UPDATE: i32 = 23;
pub const SQLITE_ATTACH: i32 = 24;
pub const SQLITE_DETACH: i32 = 25;
pub const SQLITE_ALTER_TABLE: i32 = 26;
pub const SQLITE_REINDEX: i32 = 27;
pub const SQLITE_ANALYZE: i32 = 28;
pub const SQLITE_CREATE_VTABLE: i32 = 29;
pub const SQLITE_DROP_VTABLE: i32 = 30;
pub const SQLITE_FUNCTION: i32 = 31;
pub const SQLITE_SAVEPOINT: i32 = 32;
pub const SQLITE_COPY: i32 = 0;
pub const SQLITE_RECURSIVE: i32 = 33;
pub const SQLITE_TRACE_STMT: i32 = 1;
pub const SQLITE_TRACE_PROFILE: i32 = 2;
pub const SQLITE_TRACE_ROW: i32 = 4;
pub const SQLITE_TRACE_CLOSE: i32 = 8;
pub const SQLITE_LIMIT_LENGTH: i32 = 0;
```

```
pub const SQLITE_LIMIT_SQL_LENGTH: i32 = 1;
pub const SQLITE_LIMIT_COLUMN: i32 = 2;
pub const SQLITE_LIMIT_EXPR_DEPTH: i32 = 3;
pub const SQLITE_LIMIT_COMPOUND_SELECT: i32 = 4;
pub const SQLITE_LIMIT_VDBE_OP: i32 = 5;
pub const SQLITE_LIMIT_FUNCTION_ARG: i32 = 6;
pub const SQLITE_LIMIT_ATTACHED: i32 = 7;
pub const SQLITE_LIMIT_LIKE_PATTERN_LENGTH: i32 = 8;
pub const SQLITE_LIMIT_VARIABLE_NUMBER: i32 = 9;
pub const SQLITE_LIMIT_TRIGGER_DEPTH: i32 = 10;
pub const SQLITE_LIMIT_WORKER_THREADS: i32 = 11;
pub const SQLITE_PREPARE_PERSISTENT: i32 = 1;
pub const SQLITE_PREPARE_NORMALIZE: i32 = 2;
pub const SQLITE_PREPARE_NO_VTAB: i32 = 4;
pub const SQLITE_INTEGER: i32 = 1;
pub const SQLITE_FLOAT: i32 = 2;
pub const SQLITE_BLOB: i32 = 4;
pub const SQLITE_NULL: i32 = 5;
pub const SQLITE_TEXT: i32 = 3;
pub const SQLITE3_TEXT: i32 = 3;
pub const SQLITE_UTF8: i32 = 1;
pub const SQLITE_UTF16LE: i32 = 2;
pub const SQLITE_UTF16BE: i32 = 3;
pub const SQLITE_UTF16: i32 = 4;
pub const SQLITE_ANY: i32 = 5;
pub const SQLITE_UTF16_ALIGNED: i32 = 8;
pub const SQLITE_DETERMINISTIC: i32 = 2048;
pub const SQLITE_DIRECTONLY: i32 = 524288;
pub const SQLITE_SUBTYPE: i32 = 1048576;
pub const SQLITE_INNOCUOUS: i32 = 2097152;
pub const SQLITE_WIN32_DATA_DIRECTORY_TYPE: i32 = 1;
pub const SQLITE_WIN32_TEMP_DIRECTORY_TYPE: i32 = 2;
pub const SQLITE_TXN_NONE: i32 = 0;
pub const SQLITE_TXN_READ: i32 = 1;
pub const SQLITE_TXN_WRITE: i32 = 2;
pub const SQLITE_INDEX_SCAN_UNIQUE: i32 = 1;
pub const SQLITE_INDEX_CONSTRAINT_EQ: i32 = 2;
pub const SQLITE_INDEX_CONSTRAINT_GT: i32 = 4;
pub const SQLITE_INDEX_CONSTRAINT_LE: i32 = 8;
pub const SQLITE_INDEX_CONSTRAINT_LT: i32 = 16;
pub const SQLITE_INDEX_CONSTRAINT_GE: i32 = 32;
pub const SQLITE_INDEX_CONSTRAINT_MATCH: i32 = 64;
pub const SQLITE_INDEX_CONSTRAINT_LIKE: i32 = 65;
pub const SQLITE_INDEX_CONSTRAINT_GLOB: i32 = 66;
pub const SQLITE_INDEX_CONSTRAINT_REGEXP: i32 = 67;
pub const SQLITE_INDEX_CONSTRAINT_NE: i32 = 68;
pub const SQLITE_INDEX_CONSTRAINT_ISNOT: i32 = 69;
pub const SQLITE_INDEX_CONSTRAINT_ISNOTNULL: i32 = 70;
pub const SQLITE_INDEX_CONSTRAINT_ISNULL: i32 = 71;
pub const SQLITE_INDEX_CONSTRAINT_IS: i32 = 72;
pub const SQLITE_INDEX_CONSTRAINT_LIMIT: i32 = 73;
pub const SQLITE_INDEX_CONSTRAINT_OFFSET: i32 = 74;
```

```
pub const SQLITE_INDEX_CONSTRAINT_FUNCTION: i32 = 150;
pub const SQLITE_MUTEX_FAST: i32 = 0;
pub const SQLITE_MUTEX_RECURSIVE: i32 = 1;
pub const SQLITE_MUTEX_STATIC_MAIN: i32 = 2;
pub const SQLITE_MUTEX_STATIC_MEM: i32 = 3;
pub const SQLITE_MUTEX_STATIC_MEM2: i32 = 4;
pub const SQLITE_MUTEX_STATIC_OPEN: i32 = 4;
pub const SQLITE_MUTEX_STATIC_PRNG: i32 = 5;
pub const SQLITE_MUTEX_STATIC_LRU: i32 = 6;
pub const SQLITE_MUTEX_STATIC_LRU2: i32 = 7;
pub const SQLITE_MUTEX_STATIC_PMEM: i32 = 7;
pub const SQLITE_MUTEX_STATIC_APP1: i32 = 8;
pub const SQLITE_MUTEX_STATIC_APP2: i32 = 9;
pub const SQLITE_MUTEX_STATIC_APP3: i32 = 10;
pub const SQLITE_MUTEX_STATIC_VFS1: i32 = 11;
pub const SQLITE_MUTEX_STATIC_VFS2: i32 = 12;
pub const SQLITE_MUTEX_STATIC_VFS3: i32 = 13;
pub const SQLITE_MUTEX_STATIC_MASTER: i32 = 2;
pub const SQLITE_TESTCTRL_FIRST: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_SAVE: i32 = 5;
pub const SQLITE_TESTCTRL_PRNG_RESTORE: i32 = 6;
pub const SQLITE_TESTCTRL_PRNG_RESET: i32 = 7;
pub const SQLITE_TESTCTRL_BITVEC_TEST: i32 = 8;
pub const SQLITE_TESTCTRL_FAULT_INSTALL: i32 = 9;
pub const SQLITE_TESTCTRL_BENIGN_MALLOC_HOOKS: i32 = 10;
pub const SQLITE_TESTCTRL_PENDING_BYTE: i32 = 11;
pub const SQLITE_TESTCTRL_ASSERT: i32 = 12;
pub const SQLITE_TESTCTRL_ALWAYS: i32 = 13;
pub const SQLITE_TESTCTRL_RESERVE: i32 = 14;
pub const SQLITE_TESTCTRL_OPTIMIZATIONS: i32 = 15;
pub const SQLITE_TESTCTRL_ISKEYWORD: i32 = 16;
pub const SQLITE_TESTCTRL_SCRATCHMALLOC: i32 = 17;
pub const SQLITE_TESTCTRL_INTERNAL_FUNCTIONS: i32 = 17;
pub const SQLITE_TESTCTRL_LOCALTIME_FAULT: i32 = 18;
pub const SQLITE_TESTCTRL_EXPLAIN_STMT: i32 = 19;
pub const SQLITE_TESTCTRL_ONCE_RESET_THRESHOLD: i32 = 19;
pub const SQLITE_TESTCTRL_NEVER_CORRUPT: i32 = 20;
pub const SQLITE_TESTCTRL_VDBE_COVERAGE: i32 = 21;
pub const SQLITE_TESTCTRL_BYTEORDER: i32 = 22;
pub const SQLITE_TESTCTRL_ISINIT: i32 = 23;
pub const SQLITE_TESTCTRL_SORTER_MMAP: i32 = 24;
pub const SQLITE_TESTCTRL_IMPOSTER: i32 = 25;
pub const SQLITE_TESTCTRL_PARSER_COVERAGE: i32 = 26;
pub const SQLITE_TESTCTRL_RESULT_INTREAL: i32 = 27;
pub const SQLITE_TESTCTRL_PRNG_SEED: i32 = 28;
pub const SQLITE_TESTCTRL_EXTRA_SCHEMA_CHECKS: i32 = 29;
pub const SQLITE_TESTCTRL_SEEK_COUNT: i32 = 30;
pub const SQLITE_TESTCTRL_TRACEFLAGS: i32 = 31;
pub const SQLITE_TESTCTRL_TUNE: i32 = 32;
pub const SQLITE_TESTCTRL_LOGEST: i32 = 33;
pub const SQLITE_TESTCTRL_LAST: i32 = 33;
pub const SQLITE_STATUS_MEMORY_USED: i32 = 0;
```

```
pub const SQLITE_STATUS_PAGECACHE_USED: i32 = 1;
pub const SQLITE_STATUS_PAGECACHE_OVERFLOW: i32 = 2;
pub const SQLITE_STATUS_SCRATCH_USED: i32 = 3;
pub const SQLITE_STATUS_SCRATCH_OVERFLOW: i32 = 4;
pub const SQLITE_STATUS_MALLOC_SIZE: i32 = 5;
pub const SQLITE_STATUS_PARSER_STACK: i32 = 6;
pub const SQLITE_STATUS_PAGECACHE_SIZE: i32 = 7;
pub const SQLITE_STATUS_SCRATCH_SIZE: i32 = 8;
pub const SQLITE_STATUS_MALLOC_COUNT: i32 = 9;
pub const SQLITE_DBSTATUS_LOOKASIDE_USED: i32 = 0;
pub const SQLITE_DBSTATUS_CACHE_USED: i32 = 1;
pub const SQLITE_DBSTATUS_SCHEMA_USED: i32 = 2;
pub const SQLITE_DBSTATUS_STMT_USED: i32 = 3;
pub const SQLITE_DBSTATUS_LOOKASIDE_HIT: i32 = 4;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_SIZE: i32 = 5;
pub const SQLITE_DBSTATUS_LOOKASIDE_MISS_FULL: i32 = 6;
pub const SQLITE_DBSTATUS_CACHE_HIT: i32 = 7;
pub const SQLITE_DBSTATUS_CACHE_MISS: i32 = 8;
pub const SQLITE_DBSTATUS_CACHE_WRITE: i32 = 9;
pub const SQLITE_DBSTATUS_DEFERRED_FKS: i32 = 10;
pub const SQLITE_DBSTATUS_CACHE_USED_SHARED: i32 = 11;
pub const SQLITE_DBSTATUS_CACHE_SPILL: i32 = 12;
pub const SQLITE_DBSTATUS_MAX: i32 = 12;
pub const SQLITE_STMTSTATUS_FULLSCAN_STEP: i32 = 1;
pub const SQLITE_STMTSTATUS_SORT: i32 = 2;
pub const SQLITE_STMTSTATUS_AUTOINDEX: i32 = 3;
pub const SQLITE_STMTSTATUS_VM_STEP: i32 = 4;
pub const SQLITE_STMTSTATUS_REPREPARE: i32 = 5;
pub const SQLITE_STMTSTATUS_RUN: i32 = 6;
pub const SQLITE_STMTSTATUS_FILTER_MISS: i32 = 7;
pub const SQLITE_STMTSTATUS_FILTER_HIT: i32 = 8;
pub const SQLITE_STMTSTATUS_MEMUSED: i32 = 99;
pub const SQLITE_CHECKPOINT_PASSIVE: i32 = 0;
pub const SQLITE_CHECKPOINT_FULL: i32 = 1;
pub const SQLITE_CHECKPOINT_RESTART: i32 = 2;
pub const SQLITE_CHECKPOINT_TRUNCATE: i32 = 3;
pub const SQLITE_VTAB_CONSTRAINT_SUPPORT: i32 = 1;
pub const SQLITE_VTAB_INNOCUOUS: i32 = 2;
pub const SQLITE_VTAB_DIRECTONLY: i32 = 3;
pub const SQLITE_ROLLBACK: i32 = 1;
pub const SQLITE_FAIL: i32 = 3;
pub const SQLITE_REPLACE: i32 = 5;
pub const SQLITE_SCANSTAT_NLOOP: i32 = 0;
pub const SQLITE_SCANSTAT_NVISIT: i32 = 1;
pub const SQLITE_SCANSTAT_EST: i32 = 2;
pub const SQLITE_SCANSTAT_NAME: i32 = 3;
pub const SQLITE_SCANSTAT_EXPLAIN: i32 = 4;
pub const SQLITE_SCANSTAT_SELECTID: i32 = 5;
pub const SQLITE_SCANSTAT_PARENTID: i32 = 6;
pub const SQLITE_SCANSTAT_NCYCLE: i32 = 7;
pub const SQLITE_SCANSTAT_COMPLEX: i32 = 1;
pub const SQLITE_SERIALIZE_NOCOPY: i32 = 1;
```

```
pub const SQLITE_DESERIALIZE_FREEONCLOSE: i32 = 1;
pub const SQLITE_DESERIALIZE_RESIZEABLE: i32 = 2;
pub const SQLITE_DESERIALIZE_READONLY: i32 = 4;
pub const NOT_WITHIN: i32 = 0;
pub const PARTLY_WITHIN: i32 = 1;
pub const FULLY_WITHIN: i32 = 2;
pub const __SQLITESESSION_H_: i32 = 1;
pub const SQLITE_SESSION_OBJCONFIG_SIZE: i32 = 1;
pub const SQLITE_CHANGESETSTART_INVERT: i32 = 2;
pub const SQLITE_CHANGESETAPPLY_NOSAVEPOINT: i32 = 1;
pub const SQLITE_CHANGESETAPPLY_INVERT: i32 = 2;
pub const SQLITE_CHANGESET_DATA: i32 = 1;
pub const SQLITE_CHANGESET_NOTFOUND: i32 = 2;
pub const SQLITE_CHANGESET_CONFLICT: i32 = 3;
pub const SQLITE_CHANGESET_CONSTRAINT: i32 = 4;
pub const SQLITE_CHANGESET_FOREIGN_KEY: i32 = 5;
pub const SQLITE_CHANGESET_OMIT: i32 = 0;
pub const SQLITE_CHANGESET_REPLACE: i32 = 1;
pub const SQLITE_CHANGESET_ABORT: i32 = 2;
pub const SQLITE_SESSION_CONFIG_STRMSIZE: i32 = 1;
pub const FTS5_TOKENIZE_QUERY: i32 = 1;
pub const FTS5_TOKENIZE_PREFIX: i32 = 2;
pub const FTS5_TOKENIZE_DOCUMENT: i32 = 4;
pub const FTS5_TOKENIZE_AUX: i32 = 8;
pub const FTS5_TOKEN_COLOCATED: i32 = 1;
extern "C" {
    pub static sqlite3_version: [::std::os::raw::c_char; 0usize];
}
extern "C" {
    pub fn sqlite3_libversion() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_sourceid() -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_libversion_number() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_used(
        zOptName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_compileoption_get(N: ::std::os::raw::c_int) -> *const ::
}
extern "C" {
    pub fn sqlite3_threadsafe() -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3 {
    _unused: [u8; 0],
```

```
}
pub type sqlite_int64 = ::std::os::raw::c_longlong;
pub type sqlite_uint64 = ::std::os::raw::c_ulonglong;
pub type sqlite3_int64 = sqlite_int64;
pub type sqlite3_uint64 = sqlite_uint64;
extern "C" {
    pub fn sqlite3_close(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_close_v2(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
pub type sqlite3_callback = ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut ::std::os::raw::c_char,
        arg4: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>;
extern "C" {
    pub fn sqlite3_exec(
        arg1: *mut sqlite3,
        sql: *const ::std::os::raw::c_char,
        callback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut ::std::os::raw::c_char,
                arg4: *mut *mut ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        arg2: *mut ::std::os::raw::c_void,
        errmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_file {
    pub pMethods: *const sqlite3_io_methods,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_io_methods {
    pub iVersion: ::std::os::raw::c_int,
    pub xClose: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xRead: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            arg2: *mut ::std::os::raw::c_void,
            iAmt: ::std::os::raw::c_int,
```

```
        iOfst: sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xWrite: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: *const ::std::os::raw::c_void,
        iAmt: ::std::os::raw::c_int,
        iOfst: sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xTruncate: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_file, size: sqlite3_int64)
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pSize: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xUnlock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCheckReservedLock: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        pResOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xFileControl: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_file,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int,
>,
pub xSectorSize: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xDeviceCharacteristics: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_file) -> ::std::os::raw::c_
    >,
    pub xShmMap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iPg: ::std::os::raw::c_int,
            pgsz: ::std::os::raw::c_int,
            arg2: ::std::os::raw::c_int,
            arg3: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmLock: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            offset: ::std::os::raw::c_int,
            n: ::std::os::raw::c_int,
            flags: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShmBarrier: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xShmUnmap: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            deleteFlag: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            iAmt: ::std::os::raw::c_int,
            pp: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xUnfetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_file,
            iOfst: sqlite3_int64,
            p: *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
```

```
pub struct sqlite3_api_routines {
    _unused: [u8; 0],
}
pub type sqlite3_filename = *const ::std::os::raw::c_char;
pub type sqlite3_syscall_ptr = ::std::option::Option<unsafe extern "C" fn()
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vfs {
    pub iVersion: ::std::os::raw::c_int,
    pub szOsFile: ::std::os::raw::c_int,
    pub mxPathname: ::std::os::raw::c_int,
    pub pNext: *mut sqlite3_vfs,
    pub zName: *const ::std::os::raw::c_char,
    pub pAppData: *mut ::std::os::raw::c_void,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: sqlite3_filename,
            arg2: *mut sqlite3_file,
            flags: ::std::os::raw::c_int,
            pOutFlags: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            syncDir: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xAccess: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            flags: ::std::os::raw::c_int,
            pResOut: *mut ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xFullPathname: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            nOut: ::std::os::raw::c_int,
            zOut: *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDlOpen: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zFilename: *const ::std::os::raw::c_char,
        ) -> *mut ::std::os::raw::c_void,
    >,
```

```
pub xDlError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zErrMsg: *mut ::std::os::raw::c_char,
    ),
>,
pub xDlSym: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut ::std::os::raw::c_void,
        zSymbol: *const ::std::os::raw::c_char,
    ) -> ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            arg2: *mut ::std::os::raw::c_void,
            zSymbol: *const ::std::os::raw::c_char,
        ),
    >,
>,
pub xDlClose: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut ::std::os::
>,
pub xRandomness: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        nByte: ::std::os::raw::c_int,
        zOut: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xSleep: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        microseconds: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTime: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vfs, arg2: *mut f64) -> ::s
>,
pub xGetLastError: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int,
>,
pub xCurrentTimeInt64: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vfs,
        arg2: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
```

```
    pub xSetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
            arg2: sqlite3_syscall_ptr,
        ) -> ::std::os::raw::c_int,
    >,
    pub xGetSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> sqlite3_syscall_ptr,
    >,
    pub xNextSystemCall: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_vfs,
            zName: *const ::std::os::raw::c_char,
        ) -> *const ::std::os::raw::c_char,
    >,
}
extern "C" {
    pub fn sqlite3_initialize() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_shutdown() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_init() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_os_end() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_config(arg1: ::std::os::raw::c_int, ...) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_db_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mem_methods {
    pub xMalloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut ::std::os
    >,
    pub xFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std:
    pub xRealloc: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
```

```
            arg2: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
    >,
    pub xSize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xRoundup: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> ::std::os::raw
    >,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub pAppData: *mut ::std::os::raw::c_void,
}
extern "C" {
    pub fn sqlite3_extended_result_codes(
        arg1: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_last_insert_rowid(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_set_last_insert_rowid(arg1: *mut sqlite3, arg2: sqlite3_
}
extern "C" {
    pub fn sqlite3_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_total_changes(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_total_changes64(arg1: *mut sqlite3) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_interrupt(arg1: *mut sqlite3);
}
extern "C" {
    pub fn sqlite3_is_interrupted(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_complete(sql: *const ::std::os::raw::c_char) -> ::std::o
}
extern "C" {
    pub fn sqlite3_complete16(sql: *const ::std::os::raw::c_void) -> ::std:
}
extern "C" {
```

```
    pub fn sqlite3_busy_handler(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_busy_timeout(
        arg1: *mut sqlite3,
        ms: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_table(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        pazResult: *mut *mut *mut ::std::os::raw::c_char,
        pnRow: *mut ::std::os::raw::c_int,
        pnColumn: *mut ::std::os::raw::c_int,
        pzErrmsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_free_table(result: *mut *mut ::std::os::raw::c_char);
}
extern "C" {
    pub fn sqlite3_mprintf(arg1: *const ::std::os::raw::c_char, ...)
        -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_snprintf(
        arg1: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_char,
        arg3: *const ::std::os::raw::c_char,
        ...
    ) -> *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_malloc(arg1: ::std::os::raw::c_int) -> *mut ::std::os::r
}
extern "C" {
    pub fn sqlite3_malloc64(arg1: sqlite3_uint64) -> *mut ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_realloc(
        arg1: *mut ::std::os::raw::c_void,
        arg2: ::std::os::raw::c_int,
```

```
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_realloc64(
        arg1: *mut ::std::os::raw::c_void,
        arg2: sqlite3_uint64,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_free(arg1: *mut ::std::os::raw::c_void);
}
extern "C" {
    pub fn sqlite3_msize(arg1: *mut ::std::os::raw::c_void) -> sqlite3_uint
}
extern "C" {
    pub fn sqlite3_memory_used() -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_memory_highwater(resetFlag: ::std::os::raw::c_int) -> sq
}
extern "C" {
    pub fn sqlite3_randomness(N: ::std::os::raw::c_int, P: *mut ::std::os::
}
extern "C" {
    pub fn sqlite3_set_authorizer(
        arg1: *mut sqlite3,
        xAuth: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: *const ::std::os::raw::c_char,
                arg6: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pUserData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_trace(
        arg1: *mut sqlite3,
        xTrace: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
```

```
    pub fn sqlite3_profile(
        arg1: *mut sqlite3,
        xProfile: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: sqlite3_uint64,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_trace_v2(
        arg1: *mut sqlite3,
        uMask: ::std::os::raw::c_uint,
        xCallback: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: ::std::os::raw::c_uint,
                arg2: *mut ::std::os::raw::c_void,
                arg3: *mut ::std::os::raw::c_void,
                arg4: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_progress_handler(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg4: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3_open(
        filename: *const ::std::os::raw::c_char,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open16(
        filename: *const ::std::os::raw::c_void,
        ppDb: *mut *mut sqlite3,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_open_v2(
        filename: *const ::std::os::raw::c_char,
```

```
        ppDb: *mut *mut sqlite3,
        flags: ::std::os::raw::c_int,
        zVfs: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_parameter(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_uri_boolean(
        z: sqlite3_filename,
        zParam: *const ::std::os::raw::c_char,
        bDefault: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_uri_int64(
        arg1: sqlite3_filename,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_int64,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_uri_key(
        z: sqlite3_filename,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_filename_database(arg1: sqlite3_filename) -> *const ::st
}
extern "C" {
    pub fn sqlite3_filename_journal(arg1: sqlite3_filename) -> *const ::std
}
extern "C" {
    pub fn sqlite3_filename_wal(arg1: sqlite3_filename) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_database_file_object(arg1: *const ::std::os::raw::c_char
}
extern "C" {
    pub fn sqlite3_create_filename(
        zDatabase: *const ::std::os::raw::c_char,
        zJournal: *const ::std::os::raw::c_char,
        zWal: *const ::std::os::raw::c_char,
        nParam: ::std::os::raw::c_int,
        azParam: *mut *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
```

```
extern "C" {
    pub fn sqlite3_free_filename(arg1: sqlite3_filename);
}
extern "C" {
    pub fn sqlite3_errcode(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_extended_errcode(db: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_errmsg(arg1: *mut sqlite3) -> *const ::std::os::raw::c_c
}
extern "C" {
    pub fn sqlite3_errmsg16(arg1: *mut sqlite3) -> *const ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_errstr(arg1: ::std::os::raw::c_int) -> *const ::std::os:
}
extern "C" {
    pub fn sqlite3_error_offset(db: *mut sqlite3) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_stmt {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_limit(
        arg1: *mut sqlite3,
        id: ::std::os::raw::c_int,
        newVal: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_prepare_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_char,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v2(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_prepare16_v3(
        db: *mut sqlite3,
        zSql: *const ::std::os::raw::c_void,
        nByte: ::std::os::raw::c_int,
        prepFlags: ::std::os::raw::c_uint,
        ppStmt: *mut *mut sqlite3_stmt,
        pzTail: *mut *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_sql(pStmt: *mut sqlite3_stmt) -> *const ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_expanded_sql(pStmt: *mut sqlite3_stmt) -> *mut ::std::os
}
extern "C" {
    pub fn sqlite3_stmt_readonly(pStmt: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_stmt_isexplain(pStmt: *mut sqlite3_stmt) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_stmt_busy(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
```

```
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_value {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_context {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_bind_blob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_blob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_double(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: f64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_int64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_null(
        arg1: *mut sqlite3_stmt,
```

```
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_void,
        arg4: ::std::os::raw::c_int,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_text64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const ::std::os::raw::c_char,
        arg4: sqlite3_uint64,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_value(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *const sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_pointer(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: *mut ::std::os::raw::c_void,
        arg4: *const ::std::os::raw::c_char,
        arg5: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        n: ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_zeroblob64(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
        arg3: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_bind_parameter_count(arg1: *mut sqlite3_stmt) -> ::std::
}
extern "C" {
    pub fn sqlite3_bind_parameter_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_bind_parameter_index(
        arg1: *mut sqlite3_stmt,
        zName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_clear_bindings(arg1: *mut sqlite3_stmt) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_column_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_column_name(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_name16(
        arg1: *mut sqlite3_stmt,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_database_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_database_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
```

```
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_table_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_table_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_origin_name(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_origin_name16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_decltype(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_column_decltype16(
        arg1: *mut sqlite3_stmt,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_step(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_data_count(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_column_blob(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_double(arg1: *mut sqlite3_stmt, iCol: ::std::os::
```

```
}
extern "C" {
    pub fn sqlite3_column_int(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_int64(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_column_text(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_column_text16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_column_value(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> *mut sqlite3_value;
}
extern "C" {
    pub fn sqlite3_column_bytes(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_bytes16(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_column_type(
        arg1: *mut sqlite3_stmt,
        iCol: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_finalize(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_
}
```

```
extern "C" {
    pub fn sqlite3_reset(pStmt: *mut sqlite3_stmt) -> ::std::os::raw::c_int
}
extern "C" {
    pub fn sqlite3_create_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_function16(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_void,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_create_function_v2(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xFunc: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_window_function(
        db: *mut sqlite3,
        zFunctionName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
        eTextRep: ::std::os::raw::c_int,
        pApp: *mut ::std::os::raw::c_void,
        xStep: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xFinal: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xValue: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqlit
        xInverse: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_context,
                arg2: ::std::os::raw::c_int,
                arg3: *mut *mut sqlite3_value,
            ),
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_aggregate_count(arg1: *mut sqlite3_context) -> ::std::os
```

```
}
extern "C" {
    pub fn sqlite3_expired(arg1: *mut sqlite3_stmt) -> ::std::os::raw::c_in
}
extern "C" {
    pub fn sqlite3_transfer_bindings(
        arg1: *mut sqlite3_stmt,
        arg2: *mut sqlite3_stmt,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_global_recover() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_thread_cleanup();
}
extern "C" {
    pub fn sqlite3_memory_alarm(
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: sqlite3_int64,
                arg3: ::std::os::raw::c_int,
            ),
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_value_blob(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_double(arg1: *mut sqlite3_value) -> f64;
}
extern "C" {
    pub fn sqlite3_value_int(arg1: *mut sqlite3_value) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_value_int64(arg1: *mut sqlite3_value) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_value_pointer(
        arg1: *mut sqlite3_value,
        arg2: *const ::std::os::raw::c_char,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_value_text(arg1: *mut sqlite3_value) -> *const ::std::os
}
extern "C" {
    pub fn sqlite3_value_text16(arg1: *mut sqlite3_value) -> *const ::std::
```

```
}
extern "C" {
    pub fn sqlite3_value_text16le(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_text16be(arg1: *mut sqlite3_value) -> *const ::std
}
extern "C" {
    pub fn sqlite3_value_bytes(arg1: *mut sqlite3_value) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_value_bytes16(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_type(arg1: *mut sqlite3_value) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_value_numeric_type(arg1: *mut sqlite3_value) -> ::std::o
}
extern "C" {
    pub fn sqlite3_value_nochange(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_frombind(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_encoding(arg1: *mut sqlite3_value) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_value_subtype(arg1: *mut sqlite3_value) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_value_dup(arg1: *const sqlite3_value) -> *mut sqlite3_va
}
extern "C" {
    pub fn sqlite3_value_free(arg1: *mut sqlite3_value);
}
extern "C" {
    pub fn sqlite3_aggregate_context(
        arg1: *mut sqlite3_context,
        nBytes: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_user_data(arg1: *mut sqlite3_context) -> *mut ::std::os:
}
extern "C" {
    pub fn sqlite3_context_db_handle(arg1: *mut sqlite3_context) -> *mut sq
}
extern "C" {
    pub fn sqlite3_get_auxdata(
        arg1: *mut sqlite3_context,
```

```
        N: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_set_auxdata(
        arg1: *mut sqlite3_context,
        N: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
pub type sqlite3_destructor_type =
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::os::raw::c
extern "C" {
    pub fn sqlite3_result_blob(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_blob64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_double(arg1: *mut sqlite3_context, arg2: f64);
}
extern "C" {
    pub fn sqlite3_result_error(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_result_error_toobig(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_error_nomem(arg1: *mut sqlite3_context);
}
```

```
extern "C" {
    pub fn sqlite3_result_error_code(arg1: *mut sqlite3_context, arg2: ::st
}
extern "C" {
    pub fn sqlite3_result_int(arg1: *mut sqlite3_context, arg2: ::std::os::
}
extern "C" {
    pub fn sqlite3_result_int64(arg1: *mut sqlite3_context, arg2: sqlite3_i
}
extern "C" {
    pub fn sqlite3_result_null(arg1: *mut sqlite3_context);
}
extern "C" {
    pub fn sqlite3_result_text(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text64(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_char,
        arg3: sqlite3_uint64,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        encoding: ::std::os::raw::c_uchar,
    );
}
extern "C" {
    pub fn sqlite3_result_text16(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16le(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_text16be(
        arg1: *mut sqlite3_context,
        arg2: *const ::std::os::raw::c_void,
        arg3: ::std::os::raw::c_int,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
```

```
}
extern "C" {
    pub fn sqlite3_result_value(arg1: *mut sqlite3_context, arg2: *mut sqli
}
extern "C" {
    pub fn sqlite3_result_pointer(
        arg1: *mut sqlite3_context,
        arg2: *mut ::std::os::raw::c_void,
        arg3: *const ::std::os::raw::c_char,
        arg4: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    );
}
extern "C" {
    pub fn sqlite3_result_zeroblob(arg1: *mut sqlite3_context, n: ::std::os
}
extern "C" {
    pub fn sqlite3_result_zeroblob64(
        arg1: *mut sqlite3_context,
        n: sqlite3_uint64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_result_subtype(arg1: *mut sqlite3_context, arg2: ::std::
}
extern "C" {
    pub fn sqlite3_create_collation(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation_v2(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
```

```
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_collation16(
        arg1: *mut sqlite3,
        zName: *const ::std::os::raw::c_void,
        eTextRep: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
        xCompare: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
                arg4: ::std::os::raw::c_int,
                arg5: *const ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_collation_needed16(
        arg1: *mut sqlite3,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                eTextRep: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_void,
            ),
        >,
    ) -> ::std::os::raw::c_int;
}
```

```
extern "C" {
    pub fn sqlite3_sleep(arg1: ::std::os::raw::c_int) -> ::std::os::raw::c_
}
extern "C" {
    pub static mut sqlite3_temp_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub static mut sqlite3_data_directory: *mut ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_win32_set_directory(
        type_: ::std::os::raw::c_ulong,
        zValue: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory8(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_win32_set_directory16(
        type_: ::std::os::raw::c_ulong,
        zValue: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_get_autocommit(arg1: *mut sqlite3) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_db_handle(arg1: *mut sqlite3_stmt) -> *mut sqlite3;
}
extern "C" {
    pub fn sqlite3_db_name(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_db_filename(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> sqlite3_filename;
}
extern "C" {
    pub fn sqlite3_db_readonly(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3_txn_state(
        arg1: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_next_stmt(pDb: *mut sqlite3, pStmt: *mut sqlite3_stmt) -
}
extern "C" {
    pub fn sqlite3_commit_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::st
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_rollback_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_autovacuum_pages(
        db: *mut sqlite3,
        arg1: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *const ::std::os::raw::c_char,
                arg3: ::std::os::raw::c_uint,
                arg4: ::std::os::raw::c_uint,
                arg5: ::std::os::raw::c_uint,
            ) -> ::std::os::raw::c_uint,
        >,
        arg2: *mut ::std::os::raw::c_void,
        arg3: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::std::
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_update_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: ::std::os::raw::c_int,
                arg3: *const ::std::os::raw::c_char,
                arg4: *const ::std::os::raw::c_char,
                arg5: sqlite3_int64,
            ),
        >,
```

```
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_enable_shared_cache(arg1: ::std::os::raw::c_int) -> ::st
}
extern "C" {
    pub fn sqlite3_release_memory(arg1: ::std::os::raw::c_int) -> ::std::os
}
extern "C" {
    pub fn sqlite3_db_release_memory(arg1: *mut sqlite3) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_soft_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_hard_heap_limit64(N: sqlite3_int64) -> sqlite3_int64;
}
extern "C" {
    pub fn sqlite3_soft_heap_limit(N: ::std::os::raw::c_int);
}
extern "C" {
    pub fn sqlite3_table_column_metadata(
        db: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        zTableName: *const ::std::os::raw::c_char,
        zColumnName: *const ::std::os::raw::c_char,
        pzDataType: *mut *const ::std::os::raw::c_char,
        pzCollSeq: *mut *const ::std::os::raw::c_char,
        pNotNull: *mut ::std::os::raw::c_int,
        pPrimaryKey: *mut ::std::os::raw::c_int,
        pAutoinc: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_load_extension(
        db: *mut sqlite3,
        zFile: *const ::std::os::raw::c_char,
        zProc: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_enable_load_extension(
        db: *mut sqlite3,
        onoff: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_cancel_auto_extension(
        xEntryPoint: ::std::option::Option<unsafe extern "C" fn()>,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_reset_auto_extension();
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_module {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xConnect: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3,
            pAux: *mut ::std::os::raw::c_void,
            argc: ::std::os::raw::c_int,
            argv: *const *const ::std::os::raw::c_char,
            ppVTab: *mut *mut sqlite3_vtab,
            arg2: *mut *mut ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xBestIndex: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: *mut sqlite3_index_info,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDisconnect: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xDestroy: ::std::option::Option<
        unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
    >,
    pub xOpen: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            ppCursor: *mut *mut sqlite3_vtab_cursor,
        ) -> ::std::os::raw::c_int,
    >,
    pub xClose: ::std::option::Option<
```

```
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xFilter: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        idxNum: ::std::os::raw::c_int,
        idxStr: *const ::std::os::raw::c_char,
        argc: ::std::os::raw::c_int,
        argv: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int,
>,
pub xNext: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xEof: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut sqlite3_vtab_cursor) -> ::std::os::
>,
pub xColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        arg2: *mut sqlite3_context,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab_cursor,
        pRowid: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xUpdate: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut sqlite3_vtab,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
        arg4: *mut sqlite3_int64,
    ) -> ::std::os::raw::c_int,
>,
pub xBegin: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xSync: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xCommit: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xRollback: ::std::option::Option<
    unsafe extern "C" fn(pVTab: *mut sqlite3_vtab) -> ::std::os::raw::c
>,
pub xFindFunction: ::std::option::Option<
    unsafe extern "C" fn(
```

```
            pVtab: *mut sqlite3_vtab,
            nArg: ::std::os::raw::c_int,
            zName: *const ::std::os::raw::c_char,
            pxFunc: *mut ::std::option::Option<
                unsafe extern "C" fn(
                    arg1: *mut sqlite3_context,
                    arg2: ::std::os::raw::c_int,
                    arg3: *mut *mut sqlite3_value,
                ),
            >,
            ppArg: *mut *mut ::std::os::raw::c_void,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRename: ::std::option::Option<
        unsafe extern "C" fn(
            pVtab: *mut sqlite3_vtab,
            zNew: *const ::std::os::raw::c_char,
        ) -> ::std::os::raw::c_int,
    >,
    pub xSavepoint: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRelease: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xRollbackTo: ::std::option::Option<
        unsafe extern "C" fn(
            pVTab: *mut sqlite3_vtab,
            arg1: ::std::os::raw::c_int,
        ) -> ::std::os::raw::c_int,
    >,
    pub xShadowName: ::std::option::Option<
        unsafe extern "C" fn(arg1: *const ::std::os::raw::c_char) -> ::std:
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_info {
    pub nConstraint: ::std::os::raw::c_int,
    pub aConstraint: *mut sqlite3_index_constraint,
    pub nOrderBy: ::std::os::raw::c_int,
    pub aOrderBy: *mut sqlite3_index_orderby,
    pub aConstraintUsage: *mut sqlite3_index_constraint_usage,
    pub idxNum: ::std::os::raw::c_int,
    pub idxStr: *mut ::std::os::raw::c_char,
    pub needToFreeIdxStr: ::std::os::raw::c_int,
```

```
    pub orderByConsumed: ::std::os::raw::c_int,
    pub estimatedCost: f64,
    pub estimatedRows: sqlite3_int64,
    pub idxFlags: ::std::os::raw::c_int,
    pub colUsed: sqlite3_uint64,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint {
    pub iColumn: ::std::os::raw::c_int,
    pub op: ::std::os::raw::c_uchar,
    pub usable: ::std::os::raw::c_uchar,
    pub iTermOffset: ::std::os::raw::c_int,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_orderby {
    pub iColumn: ::std::os::raw::c_int,
    pub desc: ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_index_constraint_usage {
    pub argvIndex: ::std::os::raw::c_int,
    pub omit: ::std::os::raw::c_uchar,
}
extern "C" {
    pub fn sqlite3_create_module(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_create_module_v2(
        db: *mut sqlite3,
        zName: *const ::std::os::raw::c_char,
        p: *const sqlite3_module,
        pClientData: *mut ::std::os::raw::c_void,
        xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_drop_modules(
        db: *mut sqlite3,
        azKeep: *mut *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab {
```

```
    pub pModule: *const sqlite3_module,
    pub nRef: ::std::os::raw::c_int,
    pub zErrMsg: *mut ::std::os::raw::c_char,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_vtab_cursor {
    pub pVtab: *mut sqlite3_vtab,
}
extern "C" {
    pub fn sqlite3_declare_vtab(
        arg1: *mut sqlite3,
        zSQL: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_overload_function(
        arg1: *mut sqlite3,
        zFuncName: *const ::std::os::raw::c_char,
        nArg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_blob {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_blob_open(
        arg1: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        zTable: *const ::std::os::raw::c_char,
        zColumn: *const ::std::os::raw::c_char,
        iRow: sqlite3_int64,
        flags: ::std::os::raw::c_int,
        ppBlob: *mut *mut sqlite3_blob,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_reopen(
        arg1: *mut sqlite3_blob,
        arg2: sqlite3_int64,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_close(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_bytes(arg1: *mut sqlite3_blob) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_blob_read(
```

```
        arg1: *mut sqlite3_blob,
        Z: *mut ::std::os::raw::c_void,
        N: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_blob_write(
        arg1: *mut sqlite3_blob,
        z: *const ::std::os::raw::c_void,
        n: ::std::os::raw::c_int,
        iOffset: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_find(zVfsName: *const ::std::os::raw::c_char) -> *mu
}
extern "C" {
    pub fn sqlite3_vfs_register(
        arg1: *mut sqlite3_vfs,
        makeDflt: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vfs_unregister(arg1: *mut sqlite3_vfs) -> ::std::os::raw
}
extern "C" {
    pub fn sqlite3_mutex_alloc(arg1: ::std::os::raw::c_int) -> *mut sqlite3
}
extern "C" {
    pub fn sqlite3_mutex_free(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_enter(arg1: *mut sqlite3_mutex);
}
extern "C" {
    pub fn sqlite3_mutex_try(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_mutex_leave(arg1: *mut sqlite3_mutex);
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_mutex_methods {
    pub xMutexInit: ::std::option::Option<unsafe extern "C" fn() -> ::std::
    pub xMutexEnd: ::std::option::Option<unsafe extern "C" fn() -> ::std::o
    pub xMutexAlloc: ::std::option::Option<
        unsafe extern "C" fn(arg1: ::std::os::raw::c_int) -> *mut sqlite3_m
    >,
    pub xMutexFree: ::std::option::Option<unsafe extern "C" fn(arg1: *mut s
    pub xMutexEnter: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexTry: ::std::option::Option<
```

```
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexLeave: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    pub xMutexHeld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
    pub xMutexNotheld: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_mutex) -> ::std::os::raw::c
    >,
}
extern "C" {
    pub fn sqlite3_mutex_held(arg1: *mut sqlite3_mutex) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_mutex_notheld(arg1: *mut sqlite3_mutex) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_db_mutex(arg1: *mut sqlite3) -> *mut sqlite3_mutex;
}
extern "C" {
    pub fn sqlite3_file_control(
        arg1: *mut sqlite3,
        zDbName: *const ::std::os::raw::c_char,
        op: ::std::os::raw::c_int,
        arg2: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_test_control(op: ::std::os::raw::c_int, ...) -> ::std::o
}
extern "C" {
    pub fn sqlite3_keyword_count() -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_name(
        arg1: ::std::os::raw::c_int,
        arg2: *mut *const ::std::os::raw::c_char,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_keyword_check(
        arg1: *const ::std::os::raw::c_char,
        arg2: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_str {
    _unused: [u8; 0],
}
extern "C" {
```

```
    pub fn sqlite3_str_new(arg1: *mut sqlite3) -> *mut sqlite3_str;
}
extern "C" {
    pub fn sqlite3_str_finish(arg1: *mut sqlite3_str) -> *mut ::std::os::ra
}
extern "C" {
    pub fn sqlite3_str_appendf(arg1: *mut sqlite3_str, zFormat: *const ::st
}
extern "C" {
    pub fn sqlite3_str_append(
        arg1: *mut sqlite3_str,
        zIn: *const ::std::os::raw::c_char,
        N: ::std::os::raw::c_int,
    );
}
extern "C" {
    pub fn sqlite3_str_appendall(arg1: *mut sqlite3_str, zIn: *const ::std:
}
extern "C" {
    pub fn sqlite3_str_appendchar(
        arg1: *mut sqlite3_str,
        N: ::std::os::raw::c_int,
        C: ::std::os::raw::c_char,
    );
}
extern "C" {
    pub fn sqlite3_str_reset(arg1: *mut sqlite3_str);
}
extern "C" {
    pub fn sqlite3_str_errcode(arg1: *mut sqlite3_str) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_str_length(arg1: *mut sqlite3_str) -> ::std::os::raw::c_
}
extern "C" {
    pub fn sqlite3_str_value(arg1: *mut sqlite3_str) -> *mut ::std::os::raw
}
extern "C" {
    pub fn sqlite3_status(
        op: ::std::os::raw::c_int,
        pCurrent: *mut ::std::os::raw::c_int,
        pHighwater: *mut ::std::os::raw::c_int,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_status64(
        op: ::std::os::raw::c_int,
        pCurrent: *mut sqlite3_int64,
        pHighwater: *mut sqlite3_int64,
        resetFlag: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
```

```
}
extern "C" {
    pub fn sqlite3_db_status(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        pCur: *mut ::std::os::raw::c_int,
        pHiwtr: *mut ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_status(
        arg1: *mut sqlite3_stmt,
        op: ::std::os::raw::c_int,
        resetFlg: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_page {
    pub pBuf: *mut ::std::os::raw::c_void,
    pub pExtra: *mut ::std::os::raw::c_void,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods2 {
    pub iVersion: ::std::os::raw::c_int,
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            szExtra: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
```

```
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache_page,
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut sqlite3_pcache_page,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
    pub xShrink: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sqli
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_pcache_methods {
    pub pArg: *mut ::std::os::raw::c_void,
    pub xInit: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut ::std::os::raw::c_void) -> ::std::o
    >,
    pub xShutdown: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            szPage: ::std::os::raw::c_int,
            bPurgeable: ::std::os::raw::c_int,
        ) -> *mut sqlite3_pcache,
    >,
    pub xCachesize: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, nCachesize: ::std::
    >,
    pub xPagecount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache) -> ::std::os::raw::
    >,
    pub xFetch: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            key: ::std::os::raw::c_uint,
            createFlag: ::std::os::raw::c_int,
        ) -> *mut ::std::os::raw::c_void,
```

```
    >,
    pub xUnpin: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            discard: ::std::os::raw::c_int,
        ),
    >,
    pub xRekey: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut sqlite3_pcache,
            arg2: *mut ::std::os::raw::c_void,
            oldKey: ::std::os::raw::c_uint,
            newKey: ::std::os::raw::c_uint,
        ),
    >,
    pub xTruncate: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut sqlite3_pcache, iLimit: ::std::os::
    >,
    pub xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut sql
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_backup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3_backup_init(
        pDest: *mut sqlite3,
        zDestName: *const ::std::os::raw::c_char,
        pSource: *mut sqlite3,
        zSourceName: *const ::std::os::raw::c_char,
    ) -> *mut sqlite3_backup;
}
extern "C" {
    pub fn sqlite3_backup_step(
        p: *mut sqlite3_backup,
        nPage: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_backup_finish(p: *mut sqlite3_backup) -> ::std::os::raw:
}
extern "C" {
    pub fn sqlite3_backup_remaining(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_backup_pagecount(p: *mut sqlite3_backup) -> ::std::os::r
}
extern "C" {
    pub fn sqlite3_unlock_notify(
        pBlocked: *mut sqlite3,
```

```
        xNotify: ::std::option::Option<
            unsafe extern "C" fn(
                apArg: *mut *mut ::std::os::raw::c_void,
                nArg: ::std::os::raw::c_int,
            ),
        >,
        pNotifyArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stricmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strnicmp(
        arg1: *const ::std::os::raw::c_char,
        arg2: *const ::std::os::raw::c_char,
        arg3: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strglob(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_strlike(
        zGlob: *const ::std::os::raw::c_char,
        zStr: *const ::std::os::raw::c_char,
        cEsc: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_log(
        iErrCode: ::std::os::raw::c_int,
        zFormat: *const ::std::os::raw::c_char,
        ...
    );
}
extern "C" {
    pub fn sqlite3_wal_hook(
        arg1: *mut sqlite3,
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut ::std::os::raw::c_void,
                arg2: *mut sqlite3,
                arg3: *const ::std::os::raw::c_char,
                arg4: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
```

```
        >,
        arg3: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_wal_autocheckpoint(
        db: *mut sqlite3,
        N: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_wal_checkpoint_v2(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
        eMode: ::std::os::raw::c_int,
        pnLog: *mut ::std::os::raw::c_int,
        pnCkpt: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_config(
        arg1: *mut sqlite3,
        op: ::std::os::raw::c_int,
        ...
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_on_conflict(arg1: *mut sqlite3) -> ::std::os::raw::
}
extern "C" {
    pub fn sqlite3_vtab_nochange(arg1: *mut sqlite3_context) -> ::std::os::
}
extern "C" {
    pub fn sqlite3_vtab_collation(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
    ) -> *const ::std::os::raw::c_char;
}
extern "C" {
    pub fn sqlite3_vtab_distinct(arg1: *mut sqlite3_index_info) -> ::std::o
}
extern "C" {
    pub fn sqlite3_vtab_in(
        arg1: *mut sqlite3_index_info,
        iCons: ::std::os::raw::c_int,
        bHandle: ::std::os::raw::c_int,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_first(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_in_next(
        pVal: *mut sqlite3_value,
        ppOut: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_vtab_rhs_value(
        arg1: *mut sqlite3_index_info,
        arg2: ::std::os::raw::c_int,
        ppVal: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_v2(
        pStmt: *mut sqlite3_stmt,
        idx: ::std::os::raw::c_int,
        iScanStatusOp: ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_stmt_scanstatus_reset(arg1: *mut sqlite3_stmt);
}
extern "C" {
    pub fn sqlite3_db_cacheflush(arg1: *mut sqlite3) -> ::std::os::raw::c_i
}
extern "C" {
    pub fn sqlite3_preupdate_hook(
        db: *mut sqlite3,
        xPreUpdate: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                db: *mut sqlite3,
                op: ::std::os::raw::c_int,
```

```
                zDb: *const ::std::os::raw::c_char,
                zName: *const ::std::os::raw::c_char,
                iKey1: sqlite3_int64,
                iKey2: sqlite3_int64,
            ),
        >,
        arg1: *mut ::std::os::raw::c_void,
    ) -> *mut ::std::os::raw::c_void;
}
extern "C" {
    pub fn sqlite3_preupdate_old(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_count(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_depth(arg1: *mut sqlite3) -> ::std::os::raw::c
}
extern "C" {
    pub fn sqlite3_preupdate_new(
        arg1: *mut sqlite3,
        arg2: ::std::os::raw::c_int,
        arg3: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_preupdate_blobwrite(arg1: *mut sqlite3) -> ::std::os::ra
}
extern "C" {
    pub fn sqlite3_system_errno(arg1: *mut sqlite3) -> ::std::os::raw::c_in
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_snapshot {
    pub hidden: [::std::os::raw::c_uchar; 48usize],
}
extern "C" {
    pub fn sqlite3_snapshot_get(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        ppSnapshot: *mut *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_open(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pSnapshot: *mut sqlite3_snapshot,
```

```
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_free(arg1: *mut sqlite3_snapshot);
}
extern "C" {
    pub fn sqlite3_snapshot_cmp(
        p1: *mut sqlite3_snapshot,
        p2: *mut sqlite3_snapshot,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_snapshot_recover(
        db: *mut sqlite3,
        zDb: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3_serialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        piSize: *mut sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> *mut ::std::os::raw::c_uchar;
}
extern "C" {
    pub fn sqlite3_deserialize(
        db: *mut sqlite3,
        zSchema: *const ::std::os::raw::c_char,
        pData: *mut ::std::os::raw::c_uchar,
        szDb: sqlite3_int64,
        szBuf: sqlite3_int64,
        mFlags: ::std::os::raw::c_uint,
    ) -> ::std::os::raw::c_int;
}
pub type sqlite3_rtree_dbl = f64;
extern "C" {
    pub fn sqlite3_rtree_geometry_callback(
        db: *mut sqlite3,
        zGeom: *const ::std::os::raw::c_char,
        xGeom: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *mut sqlite3_rtree_geometry,
                arg2: ::std::os::raw::c_int,
                arg3: *mut sqlite3_rtree_dbl,
                arg4: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pContext: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
```

```
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_geometry {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
}
extern "C" {
    pub fn sqlite3_rtree_query_callback(
        db: *mut sqlite3,
        zQueryFunc: *const ::std::os::raw::c_char,
        xQueryFunc: ::std::option::Option<
            unsafe extern "C" fn(arg1: *mut sqlite3_rtree_query_info) -> ::
        >,
        pContext: *mut ::std::os::raw::c_void,
        xDestructor: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rtree_query_info {
    pub pContext: *mut ::std::os::raw::c_void,
    pub nParam: ::std::os::raw::c_int,
    pub aParam: *mut sqlite3_rtree_dbl,
    pub pUser: *mut ::std::os::raw::c_void,
    pub xDelUser: ::std::option::Option<unsafe extern "C" fn(arg1: *mut ::s
    pub aCoord: *mut sqlite3_rtree_dbl,
    pub anQueue: *mut ::std::os::raw::c_uint,
    pub nCoord: ::std::os::raw::c_int,
    pub iLevel: ::std::os::raw::c_int,
    pub mxLevel: ::std::os::raw::c_int,
    pub iRowid: sqlite3_int64,
    pub rParentScore: sqlite3_rtree_dbl,
    pub eParentWithin: ::std::os::raw::c_int,
    pub eWithin: ::std::os::raw::c_int,
    pub rScore: sqlite3_rtree_dbl,
    pub apSqlParam: *mut *mut sqlite3_value,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_session {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changeset_iter {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3session_create(
        db: *mut sqlite3,
```

```
        zDb: *const ::std::os::raw::c_char,
        ppSession: *mut *mut sqlite3_session,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_delete(pSession: *mut sqlite3_session);
}
extern "C" {
    pub fn sqlite3session_object_config(
        arg1: *mut sqlite3_session,
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_enable(
        pSession: *mut sqlite3_session,
        bEnable: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_indirect(
        pSession: *mut sqlite3_session,
        bIndirect: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_attach(
        pSession: *mut sqlite3_session,
        zTab: *const ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_table_filter(
        pSession: *mut sqlite3_session,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    );
}
extern "C" {
    pub fn sqlite3session_changeset(
        pSession: *mut sqlite3_session,
        pnChangeset: *mut ::std::os::raw::c_int,
        ppChangeset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3session_changeset_size(pSession: *mut sqlite3_session) ->
}
extern "C" {
    pub fn sqlite3session_diff(
        pSession: *mut sqlite3_session,
        zFromDb: *const ::std::os::raw::c_char,
        zTbl: *const ::std::os::raw::c_char,
        pzErrMsg: *mut *mut ::std::os::raw::c_char,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset(
        pSession: *mut sqlite3_session,
        pnPatchset: *mut ::std::os::raw::c_int,
        ppPatchset: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_isempty(pSession: *mut sqlite3_session) -> ::std:
}
extern "C" {
    pub fn sqlite3session_memory_used(pSession: *mut sqlite3_session) -> sq
}
extern "C" {
    pub fn sqlite3changeset_start(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2(
        pp: *mut *mut sqlite3_changeset_iter,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_next(pIter: *mut sqlite3_changeset_iter) -> ::s
}
extern "C" {
    pub fn sqlite3changeset_op(
        pIter: *mut sqlite3_changeset_iter,
        pzTab: *mut *const ::std::os::raw::c_char,
        pnCol: *mut ::std::os::raw::c_int,
        pOp: *mut ::std::os::raw::c_int,
        pbIndirect: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_pk(
```

```
        pIter: *mut sqlite3_changeset_iter,
        pabPK: *mut *mut ::std::os::raw::c_uchar,
        pnCol: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_old(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_new(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_conflict(
        pIter: *mut sqlite3_changeset_iter,
        iVal: ::std::os::raw::c_int,
        ppValue: *mut *mut sqlite3_value,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_fk_conflicts(
        pIter: *mut sqlite3_changeset_iter,
        pnOut: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_finalize(pIter: *mut sqlite3_changeset_iter) ->
}
extern "C" {
    pub fn sqlite3changeset_invert(
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_concat(
        nA: ::std::os::raw::c_int,
        pA: *mut ::std::os::raw::c_void,
        nB: ::std::os::raw::c_int,
        pB: *mut ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
```

```
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_changegroup {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3changegroup_new(pp: *mut *mut sqlite3_changegroup) -> ::s
}
extern "C" {
    pub fn sqlite3changegroup_add(
        arg1: *mut sqlite3_changegroup,
        nData: ::std::os::raw::c_int,
        pData: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output(
        arg1: *mut sqlite3_changegroup,
        pnData: *mut ::std::os::raw::c_int,
        ppData: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_delete(arg1: *mut sqlite3_changegroup);
}
extern "C" {
    pub fn sqlite3changeset_apply(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2(
        db: *mut sqlite3,
        nChangeset: ::std::os::raw::c_int,
        pChangeset: *mut ::std::os::raw::c_void,
```

```
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct sqlite3_rebaser {
    _unused: [u8; 0],
}
extern "C" {
    pub fn sqlite3rebaser_create(ppNew: *mut *mut sqlite3_rebaser) -> ::std
}
extern "C" {
    pub fn sqlite3rebaser_configure(
        arg1: *mut sqlite3_rebaser,
        nRebase: ::std::os::raw::c_int,
        pRebase: *const ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase(
        arg1: *mut sqlite3_rebaser,
        nIn: ::std::os::raw::c_int,
        pIn: *const ::std::os::raw::c_void,
        pnOut: *mut ::std::os::raw::c_int,
        ppOut: *mut *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_delete(p: *mut sqlite3_rebaser);
}
extern "C" {
    pub fn sqlite3changeset_apply_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
```

```
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_apply_v2_strm(
        db: *mut sqlite3,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xFilter: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                zTab: *const ::std::os::raw::c_char,
            ) -> ::std::os::raw::c_int,
        >,
        xConflict: ::std::option::Option<
            unsafe extern "C" fn(
                pCtx: *mut ::std::os::raw::c_void,
                eConflict: ::std::os::raw::c_int,
                p: *mut sqlite3_changeset_iter,
            ) -> ::std::os::raw::c_int,
        >,
        pCtx: *mut ::std::os::raw::c_void,
        ppRebase: *mut *mut ::std::os::raw::c_void,
        pnRebase: *mut ::std::os::raw::c_int,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
```

```
    pub fn sqlite3changeset_concat_strm(
        xInputA: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInA: *mut ::std::os::raw::c_void,
        xInputB: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pInB: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_invert_strm(
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
```

```
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changeset_start_v2_strm(
        pp: *mut *mut sqlite3_changeset_iter,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        flags: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_changeset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_patchset_strm(
        pSession: *mut sqlite3_session,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_add_strm(
        arg1: *mut sqlite3_changegroup,
        xInput: ::std::option::Option<
```

```
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3changegroup_output_strm(
        arg1: *mut sqlite3_changegroup,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3rebaser_rebase_strm(
        pRebaser: *mut sqlite3_rebaser,
        xInput: ::std::option::Option<
            unsafe extern "C" fn(
                pIn: *mut ::std::os::raw::c_void,
                pData: *mut ::std::os::raw::c_void,
                pnData: *mut ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pIn: *mut ::std::os::raw::c_void,
        xOutput: ::std::option::Option<
            unsafe extern "C" fn(
                pOut: *mut ::std::os::raw::c_void,
                pData: *const ::std::os::raw::c_void,
                nData: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
        pOut: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
extern "C" {
    pub fn sqlite3session_config(
        op: ::std::os::raw::c_int,
        pArg: *mut ::std::os::raw::c_void,
    ) -> ::std::os::raw::c_int;
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Context {
```

```
    _unused: [u8; 0],
}
pub type fts5_extension_function = ::std::option::Option<
    unsafe extern "C" fn(
        pApi: *const Fts5ExtensionApi,
        pFts: *mut Fts5Context,
        pCtx: *mut sqlite3_context,
        nVal: ::std::os::raw::c_int,
        apVal: *mut *mut sqlite3_value,
    ),
>;
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5PhraseIter {
    pub a: *const ::std::os::raw::c_uchar,
    pub b: *const ::std::os::raw::c_uchar,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5ExtensionApi {
    pub iVersion: ::std::os::raw::c_int,
    pub xUserData: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> *mut ::std::os::raw
    >,
    pub xColumnCount: ::std::option::Option<
        unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
    >,
    pub xRowCount: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pnRow: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xColumnTotalSize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            iCol: ::std::os::raw::c_int,
            pnToken: *mut sqlite3_int64,
        ) -> ::std::os::raw::c_int,
    >,
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Context,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            pCtx: *mut ::std::os::raw::c_void,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    arg1: *mut ::std::os::raw::c_void,
                    arg2: ::std::os::raw::c_int,
                    arg3: *const ::std::os::raw::c_char,
                    arg4: ::std::os::raw::c_int,
```

```
                arg5: ::std::os::raw::c_int,
                arg6: ::std::os::raw::c_int,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xPhraseCount: ::std::option::Option<
    unsafe extern "C" fn(arg1: *mut Fts5Context) -> ::std::os::raw::c_i
>,
pub xPhraseSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInstCount: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pnInst: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xInst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iIdx: ::std::os::raw::c_int,
        piPhrase: *mut ::std::os::raw::c_int,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xRowid:
    ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5Context)
pub xColumnText: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pz: *mut *const ::std::os::raw::c_char,
        pn: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xColumnSize: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iCol: ::std::os::raw::c_int,
        pnToken: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xQueryPhrase: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        pUserData: *mut ::std::os::raw::c_void,
```

```
        arg2: ::std::option::Option<
            unsafe extern "C" fn(
                arg1: *const Fts5ExtensionApi,
                arg2: *mut Fts5Context,
                arg3: *mut ::std::os::raw::c_void,
            ) -> ::std::os::raw::c_int,
        >,
    ) -> ::std::os::raw::c_int,
>,
pub xSetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        pAux: *mut ::std::os::raw::c_void,
        xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
    ) -> ::std::os::raw::c_int,
>,
pub xGetAuxdata: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        bClear: ::std::os::raw::c_int,
    ) -> *mut ::std::os::raw::c_void,
>,
pub xPhraseFirst: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        arg2: *mut Fts5PhraseIter,
        arg3: *mut ::std::os::raw::c_int,
        arg4: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xPhraseNext: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        arg2: *mut Fts5PhraseIter,
        piCol: *mut ::std::os::raw::c_int,
        piOff: *mut ::std::os::raw::c_int,
    ),
>,
pub xPhraseFirstColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        iPhrase: ::std::os::raw::c_int,
        arg2: *mut Fts5PhraseIter,
        arg3: *mut ::std::os::raw::c_int,
    ) -> ::std::os::raw::c_int,
>,
pub xPhraseNextColumn: ::std::option::Option<
    unsafe extern "C" fn(
        arg1: *mut Fts5Context,
        arg2: *mut Fts5PhraseIter,
        piCol: *mut ::std::os::raw::c_int,
```

```
        ),
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct Fts5Tokenizer {
    _unused: [u8; 0],
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_tokenizer {
    pub xCreate: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut ::std::os::raw::c_void,
            azArg: *mut *const ::std::os::raw::c_char,
            nArg: ::std::os::raw::c_int,
            ppOut: *mut *mut Fts5Tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xDelete: ::std::option::Option<unsafe extern "C" fn(arg1: *mut Fts5
    pub xTokenize: ::std::option::Option<
        unsafe extern "C" fn(
            arg1: *mut Fts5Tokenizer,
            pCtx: *mut ::std::os::raw::c_void,
            flags: ::std::os::raw::c_int,
            pText: *const ::std::os::raw::c_char,
            nText: ::std::os::raw::c_int,
            xToken: ::std::option::Option<
                unsafe extern "C" fn(
                    pCtx: *mut ::std::os::raw::c_void,
                    tflags: ::std::os::raw::c_int,
                    pToken: *const ::std::os::raw::c_char,
                    nToken: ::std::os::raw::c_int,
                    iStart: ::std::os::raw::c_int,
                    iEnd: ::std::os::raw::c_int,
                ) -> ::std::os::raw::c_int,
            >,
        ) -> ::std::os::raw::c_int,
    >,
}
#[repr(C)]
#[derive(Debug, Copy, Clone)]
pub struct fts5_api {
    pub iVersion: ::std::os::raw::c_int,
    pub xCreateTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
```

```
    >,
    pub xFindTokenizer: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            ppContext: *mut *mut ::std::os::raw::c_void,
            pTokenizer: *mut fts5_tokenizer,
        ) -> ::std::os::raw::c_int,
    >,
    pub xCreateFunction: ::std::option::Option<
        unsafe extern "C" fn(
            pApi: *mut fts5_api,
            zName: *const ::std::os::raw::c_char,
            pContext: *mut ::std::os::raw::c_void,
            xFunction: fts5_extension_function,
            xDestroy: ::std::option::Option<unsafe extern "C" fn(arg1: *mut
        ) -> ::std::os::raw::c_int,
    >,
}
```

File: ./src/app/config.rs

```
use anyhow::Result;
use serde::{Deserialize, Serialize};
use std::fs;
use std::io::{self, Write};
use std::path::PathBuf;

use dirs::config_dir;

#[derive(Serialize, Deserialize, Clone)]
pub struct Config {
    pub dnac_url: String,
    pub username: String,
    pub verify_ssl: bool,
}

impl Config {
    pub fn new(dnac_url: String, username: String, verify_ssl: bool) -> Sel
        Self {
            dnac_url,
            username,
            verify_ssl,
        }
    }
}

pub fn get_config_path() -> PathBuf {
    let mut config_path = config_dir().unwrap();
    config_path.push("catalysh");
```

```
    fs::create_dir_all(&config_path).unwrap();
    config_path.push("config.yml");
    config_path
}

pub fn get_credentials_db_path() -> PathBuf {
    let mut db_path = config_dir().unwrap();
    db_path.push("catalysh");
    db_path.push("credentials.db");
    db_path
}

/// Load configuration and trigger setup if necessary
pub fn load_config() -> Result<Config> {
    let config_path = get_config_path();
    if config_path.exists() {
        let contents = fs::read_to_string(config_path)?;
        let config: Config = serde_yaml::from_str(&contents)?;
        Ok(config)
    } else {
        println!("Configuration file not found. Starting setup...");
        let config = setup_config()?;
        save_config(&config)?;
        Ok(config)
    }
}

/// Reset the configuration and credentials
pub fn reset_config() -> Result<()> {
    let config_path = get_config_path();
    let credentials_db_path = get_credentials_db_path();

    if config_path.exists() {
        fs::remove_file(config_path)?;
    }

    if credentials_db_path.exists() {
        fs::remove_file(credentials_db_path)?;
    }

    println!("Configuration files and credentials deleted.");
    Ok(())
}

/// Setup configuration by prompting the user
fn setup_config() -> Result<Config> {
    let mut dnac_url = String::new();
    let mut username = String::new();
    let mut verify_ssl_input = String::new();

    print!("Enter Cisco DNAC URL without a / at the end (e.g., https://dnac
    io::stdout().flush()?;
```

```
    io::stdin().read_line(&mut dnac_url)?;
    dnac_url = dnac_url.trim().to_string();

    print!("Enter your username: ");
    io::stdout().flush()?;
    io::stdin().read_line(&mut username)?;
    username = username.trim().to_string();

    print!("Verify SSL certificates? (y/n): ");
    io::stdout().flush()?;
    io::stdin().read_line(&mut verify_ssl_input)?;
    let verify_ssl = verify_ssl_input.trim().to_lowercase() == "y";

    println!("Configuration complete. Please proceed to store your credenti

    Ok(Config::new(dnac_url, username, verify_ssl))
}

/// Save the configuration to a file
fn save_config(config: &Config) -> Result<()> {
    let config_path = get_config_path();
    let contents = serde_yaml::to_string(config)?;
    fs::write(config_path, contents)?;
    Ok(())
}
```

File: ./src/app/update.rs

```
use std::fs;
use std::env;
use reqwest;
use serde_json::Value;
#[cfg(unix)]
use std::os::unix::fs::PermissionsExt; // Import PermissionsExt for Unix sy
#[allow(dead_code)]
pub fn update_to_latest() -> Result<(), Box<dyn std::error::Error>> {
    let repo_url = "https://api.github.com/repos/tparnell96/catalysh/releas
    let client = reqwest::blocking::Client::new();
    let response = client.get(repo_url).header("User-Agent", "Rust-App").se

    if !response.status().is_success() {
        return Err("Failed to fetch release information.".into());
    }

    let json: Value = response.json()?;
    let assets = json["assets"].as_array().ok_or("Invalid assets structure"
    let platform = if cfg!(target_os = "macos") {
        "macos"
    } else {
```

```
        "linux"
    };

    let architecture = if cfg!(target_arch = "x86_64") {
        "x86_64"
    } else if cfg!(target_arch = "aarch64") {
        "arm64"
    } else {
        "unknown"
    };

    let asset = assets.iter().find(|asset| {
        let name = asset["name"].as_str().unwrap_or("");
        name.contains(platform) && name.contains(architecture)
    }).ok_or("No compatible asset found")?;

    let download_url = asset["browser_download_url"].as_str().ok_or("Invali
    let temp_file = env::temp_dir().join(asset["name"].as_str().unwrap_or("

    println!("Downloading update...");
    let mut file = fs::File::create(&temp_file)?;
    let mut response = client.get(download_url).send()?;
    response.copy_to(&mut file)?;
    println!("Download complete: {}", temp_file.display());

    let local_bin = dirs::home_dir()
        .ok_or("Failed to determine home directory")?
        .join(".local/bin");

    if !local_bin.exists() {
        fs::create_dir_all(&local_bin)?;
    }

    let destination = local_bin.join("catalysh");

    println!("Moving binary to ~/.local/bin...");
    fs::copy(&temp_file, &destination)?;

    #[cfg(unix)]
    {
        println!("Applying executable permissions...");
        fs::set_permissions(&destination, fs::Permissions::from_mode(0o755)
    }

    println!("Update successfully applied to ~/.local/bin/catalysh.");
    Ok(())
}
```

File: ./src/app/mod.rs

```
pub mod config;
pub mod update;
```

File: ./src/bin/windows_installer.rs

```
use std::fs;
use std::env;
use std::path::PathBuf;
use std::process::Command;
use reqwest;
use serde_json::Value;

fn main() -> Result<(), Box<dyn std::error::Error>> {
    let repo_url = "https://api.github.com/repos/tparnell96/catsh/releases/
    let client = reqwest::blocking::Client::new();
    let response = client.get(repo_url).header("User-Agent", "Rust-App").se

    if !response.status().is_success() {
        return Err("Failed to fetch release information.".into());
    }

    let json: Value = response.json()?;
    let assets = json["assets"].as_array().ok_or("Invalid assets structure"

    // Download the binary file
    let binary_asset = assets.iter().find(|asset| {
        let name = asset["name"].as_str().unwrap_or("");
        name.ends_with(".exe")
    }).ok_or("No compatible binary asset found")?;
    let binary_url = binary_asset["browser_download_url"]
        .as_str()
        .ok_or("Invalid binary download URL")?;
    let binary_temp_file = env::temp_dir().join(binary_asset["name"].as_str

    println!("Downloading binary...");
    let mut file = fs::File::create(&binary_temp_file)?;
    let mut response = client.get(binary_url).send()?;
    response.copy_to(&mut file)?;
    println!("Binary download complete: {}", binary_temp_file.display());

    // Download the icon file
    let icon_asset = assets.iter().find(|asset| {
        let name = asset["name"].as_str().unwrap_or("");
        name.ends_with(".ico")
    }).ok_or("No compatible icon asset found")?;
    let icon_url = icon_asset["browser_download_url"]
        .as_str()
```

```
        .ok_or("Invalid icon download URL")?;
    let icon_temp_file = env::temp_dir().join(icon_asset["name"].as_str().u

    println!("Downloading icon...");
    let mut file = fs::File::create(&icon_temp_file)?;
    let mut response = client.get(icon_url).send()?;
    response.copy_to(&mut file)?;
    println!("Icon download complete: {}", icon_temp_file.display());

    // Move the binary to the application directory
    let app_dir = PathBuf::from(env::var("LOCALAPPDATA")?).join("catsh");
    if !app_dir.exists() {
        fs::create_dir_all(&app_dir)?;
    }
    let binary_destination = app_dir.join("catsh.exe");
    println!("Moving binary to application directory...");
    fs::copy(&binary_temp_file, &binary_destination)?;

    // Move the icon to the application directory
    let icon_destination = app_dir.join("catsh.ico");
    println!("Moving icon to application directory...");
    fs::copy(&icon_temp_file, &icon_destination)?;

    // Create the desktop shortcut
    println!("Creating desktop shortcut...");
    let desktop = dirs::desktop_dir().ok_or("Failed to locate the desktop d
    let shortcut_path = desktop.join("catsh.lnk");
    create_shortcut(&shortcut_path, &binary_destination, &icon_destination)

    println!("Update successfully applied.");
    Ok(())
}

fn create_shortcut(shortcut_path: &PathBuf, target_path: &PathBuf, icon_pat
    let output = Command::new("powershell")
        .arg("-Command")
        .arg(format!(
            r#"
            $WScript = New-Object -ComObject WScript.Shell;
            $Shortcut = $WScript.CreateShortcut('{}');
            $Shortcut.TargetPath = '{}';
            $Shortcut.IconLocation = '{}';
            $Shortcut.Save();
            "#,
            shortcut_path.display(),
            target_path.display(),
            icon_path.display()
        ))
        .output()?; // Capture output

    if !output.status.success() {
        eprintln!(
```

```
            "PowerShell failed with status: {}\nError: {}",
            output.status,
            String::from_utf8_lossy(&output.stderr)
        );
        return Err("Failed to create shortcut with icon".into());
    }

    println!(
        "PowerShell executed successfully:\n{}",
        String::from_utf8_lossy(&output.stdout)
    );

    Ok(())
}
```

File: ./src/main.rs

```
mod app;
mod helpers;
mod api;

mod commands;
mod handlers;

use commands::{Cli, route_command};
use clap_repl::reedline::{DefaultPrompt, DefaultPromptSegment, FileBackedHi
use clap_repl::ClapEditor;
use dirs::home_dir;
use std::fs;
use std::path::PathBuf;

fn get_installation_dir() -> PathBuf {
    let home = home_dir().expect("Failed to determine the user's home direc
    home.join(".catalysh")
}

fn perform_first_time_installation() -> Result<(), Box<dyn std::error::Erro
    let install_dir = get_installation_dir();

    if !install_dir.exists() {
        println!("Running first-time installation...");
        fs::create_dir_all(&install_dir)?;
        fs::write(install_dir.join("version"), "1.0.0")?;
        println!("First-time installation complete.");
    }
    Ok(())
}

#[allow(non_snake_case)]
fn main() {
```

```
    env_logger::init();
    // Initial check to confirm program is correctly installed
    if let Err(e) = perform_first_time_installation() {
        eprintln!("Error during installation: {}", e);
        return;
    }

    let prompt = DefaultPrompt {
        left_prompt: DefaultPromptSegment::Basic("catalysh".to_owned()),
        ..DefaultPrompt::default()
    };

    // Create the REPL
    let rl = ClapEditor::<Cli>::builder()
        .with_prompt(Box::new(prompt))
        .with_editor_hook(|reed| {
            reed.with_history(Box::new(
                FileBackedHistory::with_file(10000, "/tmp/catalysh-cli-hist
            ))
        })
        .build();

    rl.repl(|cli| {
        route_command(cli.command);
    });
}
```

File: ./src/api/clients/getclientenrichment.rs

```
// src/api/clients/getclientenrichment.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[serde(untagged)]
pub enum StringOrNumber {
    String(String),
    Number(u64),
}

#[derive(Debug, Deserialize)]
#[serde(untagged)]
pub enum VlanId {
    String(String),
    Number(u64),
```

```
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ClientEnrichmentResponse(pub Vec<ClientEnrichment>);

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ClientEnrichment {
    pub userDetails: Option<UserDetails>,
    pub connectedDevice: Option<Vec<EnrichmentConnectedDevice>>,
    pub issueDetails: Option<IssueDetails>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct UserDetails {
    pub id: Option<String>,
    pub connectionStatus: Option<String>,
    pub tracked: Option<String>,
    pub hostType: Option<String>,
    pub userId: Option<String>,
    pub duid: Option<String>,
    pub identifier: Option<String>,
    pub hostName: Option<String>,
    pub hostOs: Option<String>,
    pub hostVersion: Option<String>,
    pub subType: Option<String>,
    pub firmwareVersion: Option<String>,
    pub deviceVendor: Option<String>,
    pub deviceForm: Option<String>,
    pub salesCode: Option<String>,
    pub countryCode: Option<String>,
    pub lastUpdated: Option<i64>,
    pub healthScore: Option<Vec<HealthScore>>,
    pub hostMac: Option<String>,
    pub hostIpV4: Option<String>,
    pub hostIpV6: Option<Vec<String>>,
    pub authType: Option<String>,
    pub vlanId: Option<VlanId>,
    pub port: Option<String>,
    pub ssid: Option<String>,
    pub frequency: Option<String>,
    pub channel: Option<String>,
    pub apGroup: Option<String>,
    pub sgt: Option<String>,
    pub location: Option<String>,
    pub clientConnection: Option<String>,
    pub connectedDevice: Option<Vec<ConnectedDevice>>,
    pub issueCount: Option<StringOrNumber>,
    pub rssi: Option<StringOrNumber>,
    pub rssiThreshold: Option<String>,
```

```
    pub rssiIsInclude: Option<String>,
    pub avgRssi: Option<String>,
    pub snr: Option<StringOrNumber>,
    pub snrThreshold: Option<String>,
    pub snrIsInclude: Option<String>,
    pub avgSnr: Option<String>,
    pub dataRate: Option<StringOrNumber>,
    pub txBytes: Option<String>,
    pub rxBytes: Option<String>,
    pub dnsResponse: Option<String>,
    pub dnsRequest: Option<String>,
    pub onboarding: Option<Onboarding>,
    // ... other fields as needed
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct HealthScore {
    pub healthType: Option<String>,
    pub reason: Option<String>,
    pub score: Option<i32>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ConnectedDevice {
    #[serde(rename = "type")]
    pub type_field: Option<String>, // `type` is a reserved keyword in Rust
    pub name: Option<String>,
    pub mac: Option<String>,
    pub id: Option<String>,
    #[serde(rename = "ip address")]
    pub ip_address: Option<String>,
    pub mgmtIp: Option<String>,
    pub band: Option<String>,
    pub mode: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct EnrichmentConnectedDevice {
    pub deviceDetails: Option<DeviceDetails>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct DeviceDetails {
    pub family: Option<String>,
    #[serde(rename = "type")]
    pub type_field: Option<String>,
    pub location: Option<String>,
    pub errorCode: Option<String>,
```

```
    pub macAddress: Option<String>,
    pub role: Option<String>,
    pub apManagerInterfaceIp: Option<String>,
    pub associatedWlcIp: Option<String>,
    pub bootDateTime: Option<String>,
    pub collectionStatus: Option<String>,
    pub interfaceCount: Option<String>,
    pub lineCardCount: Option<String>,
    pub lineCardId: Option<String>,
    pub managementIpAddress: Option<String>,
    pub memorySize: Option<String>,
    pub platformId: Option<String>,
    pub reachabilityFailureReason: Option<String>,
    pub reachabilityStatus: Option<String>,
    pub snmpContact: Option<String>,
    pub snmpLocation: Option<String>,
    pub tunnelUdpPort: Option<String>,
    pub waasDeviceMode: Option<String>,
    pub series: Option<String>,
    pub inventoryStatusDetail: Option<String>,
    pub collectionInterval: Option<String>,
    pub serialNumber: Option<String>,
    pub softwareVersion: Option<String>,
    pub roleSource: Option<String>,
    pub hostname: Option<String>,
    pub upTime: Option<String>,
    pub lastUpdateTime: Option<i64>,
    pub errorDescription: Option<String>,
    pub locationName: Option<String>,
    pub tagCount: Option<String>,
    pub lastUpdated: Option<String>,
    pub instanceUuid: Option<String>,
    pub id: Option<String>,
    pub neighborTopology: Option<Vec<NeighborTopology>>,
    // ... other fields as needed
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct NeighborTopology {
    pub nodes: Option<Vec<TopologyNode>>,
    pub links: Option<Vec<TopologyLink>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct TopologyNode {
    pub role: Option<String>,
    pub name: Option<String>,
    pub id: Option<String>,
    pub description: Option<String>,
    pub deviceType: Option<String>,
```

```
    pub platformId: Option<String>,
    pub family: Option<String>,
    pub ip: Option<String>,
    pub softwareVersion: Option<String>,
    pub userId: Option<String>,
    pub nodeType: Option<String>,
    pub radioFrequency: Option<String>,
    pub clients: Option<f64>,
    pub count: Option<f64>,
    pub healthScore: Option<f64>,
    pub level: Option<f64>,
    pub fabricGroup: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct TopologyLink {
    pub source: Option<String>,
    pub linkStatus: Option<String>,
    pub label: Option<Vec<String>>,
    pub target: Option<String>,
    pub id: Option<String>,
    pub portUtilization: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct Onboarding {
    pub averageRunDuration: Option<String>,
    pub maxRunDuration: Option<String>,
    pub averageAssocDuration: Option<String>,
    pub maxAssocDuration: Option<String>,
    pub averageAuthDuration: Option<String>,
    pub maxAuthDuration: Option<String>,
    pub averageDhcpDuration: Option<String>,
    pub maxDhcpDuration: Option<String>,
    pub aaaServerIp: Option<String>,
    pub dhcpServerIp: Option<String>,
    pub authDoneTime: Option<i64>,
    pub assocDoneTime: Option<i64>,
    pub dhcpDoneTime: Option<i64>,
    pub latestRootCauseList: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct IssueDetails {
    pub issue: Option<Vec<Issue>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
```

```
pub struct Issue {
    pub issueId: Option<String>,
    pub issueSource: Option<String>,
    pub issueCategory: Option<String>,
    pub issueName: Option<String>,
    pub issueDescription: Option<String>,
    pub issueEntity: Option<String>,
    pub issueEntityValue: Option<String>,
    pub issueSeverity: Option<String>,
    pub issuePriority: Option<String>,
    pub issueSummary: Option<String>,
    pub issueTimestamp: Option<i64>,
    pub suggestedActions: Option<Vec<SuggestedAction>>,
    pub impactedHosts: Option<Vec<ImpactedHost>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct SuggestedAction {
    pub message: Option<String>,
    pub steps: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ImpactedHost {
    pub hostType: Option<String>,
    pub hostName: Option<String>,
    pub hostOs: Option<String>,
    pub ssid: Option<String>,
    pub connectedInterface: Option<String>,
    pub macAddress: Option<String>,
    pub failedAttempts: Option<i32>,
    pub location: Option<ImpactedHostLocation>,
    pub timestamp: Option<i64>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
pub struct ImpactedHostLocation {
    pub siteId: Option<String>,
    pub siteType: Option<String>,
    pub area: Option<String>,
    pub building: Option<String>,
    pub floor: Option<String>,
    pub apsImpacted: Option<Vec<String>>,
}

pub async fn get_client_enrichment(
    config: &Config,
    token: &Token,
    entity_type: &str,
```

```
    entity_value: &str,
    issue_category: Option<&str>,
) -> Result<ClientEnrichmentResponse> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/client-enrichment-details", con

    // Build the request with headers
    let mut req_builder = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .header("entity_type", entity_type)
        .header("entity_value", entity_value);

    if let Some(category) = issue_category {
        req_builder = req_builder.header("issueCategory", category);
    }

    let resp = req_builder.send().await?;

    if !resp.status().is_success() {
        let status = resp.status();
        let error_text = resp.text().await.unwrap_or_default();
        return Err(anyhow!(
            "Failed to retrieve client enrichment details: {} - {}",
            status,
            error_text
        ));
    }

    let enrichment_response = resp.json::<ClientEnrichmentResponse>().await
    Ok(enrichment_response)
}
```

File: ./src/api/clients/getclientdetail.rs

```
// src/api/clients/getclientdetail.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ClientDetailResponse {
```

```
    pub detail: Option<ClientDetail>,
    pub connectionInfo: Option<ConnectionInfo>,
    pub topology: Option<Topology>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ClientDetail {
    pub id: Option<String>,
    pub connectionStatus: Option<String>,
    pub hostType: Option<String>,
    pub userId: Option<String>,
    pub hostName: Option<String>,
    pub hostOs: Option<String>,
    pub hostVersion: Option<String>,
    pub subType: Option<String>,
    pub lastUpdated: Option<u64>, // Changed from Option<String> to Option<
    pub healthScore: Option<Vec<HealthScore>>,
    pub hostMac: Option<String>,
    pub hostIpV4: Option<String>,
    pub hostIpV6: Option<Vec<String>>,
    pub authType: Option<String>,
    pub vlanId: Option<i32>,
    pub vnid: Option<i32>,
    pub ssid: Option<String>,
    pub frequency: Option<String>,
    pub channel: Option<String>,
    pub apGroup: Option<String>,
    pub location: Option<String>,
    pub clientConnection: Option<String>,
    pub connectedDevice: Option<Vec<ConnectedDevice>>,
    pub issueCount: Option<i32>,
    pub rssi: Option<String>,
    pub avgRssi: Option<String>,
    pub snr: Option<String>,
    pub avgSnr: Option<String>,
    pub dataRate: Option<String>,
    pub txBytes: Option<String>,
    pub rxBytes: Option<String>,
    pub onboarding: Option<Onboarding>,
    pub clientType: Option<String>,
    pub onboardingTime: Option<u64>, // Changed from Option<String> to Opti
    pub port: Option<String>,
    pub iosCapable: Option<bool>,
    pub tracked: Option<String>,
    pub duid: Option<String>,
    pub identifier: Option<String>,
    pub firmwareVersion: Option<String>,
    pub deviceVendor: Option<String>,
    pub deviceForm: Option<String>,
    pub salesCode: Option<String>,
```

```
pub countryCode: Option<String>,
pub l3VirtualNetwork: Option<String>,
pub l2VirtualNetwork: Option<String>,
pub upnId: Option<String>,
pub upnName: Option<String>,
pub sgt: Option<String>,
pub rssiThreshold: Option<String>,
pub rssiIsInclude: Option<String>,
pub snrThreshold: Option<String>,
pub snrIsInclude: Option<String>,
pub dnsResponse: Option<String>,
pub dnsRequest: Option<String>,
pub usage: Option<f64>,
pub linkSpeed: Option<f64>,
pub linkThreshold: Option<String>,
pub remoteEndDuplexMode: Option<String>,
pub txLinkError: Option<f64>,
pub rxLinkError: Option<f64>,
pub txRate: Option<f64>,
pub rxRate: Option<f64>,
pub rxRetryPct: Option<String>,
pub versionTime: Option<i64>,
pub dot11Protocol: Option<String>,
pub slotId: Option<i32>,
pub dot11ProtocolCapability: Option<String>,
pub privateMac: Option<bool>,
pub dhcpServerIp: Option<String>,
pub aaaServerIp: Option<String>,
pub aaaServerTransaction: Option<i32>,
pub aaaServerFailedTransaction: Option<i32>,
pub aaaServerSuccessTransaction: Option<i32>,
pub aaaServerLatency: Option<f64>,
pub aaaServerMABLatency: Option<f64>,
pub aaaServerEAPLatency: Option<f64>,
pub dhcpServerTransaction: Option<i32>,
pub dhcpServerFailedTransaction: Option<i32>,
pub dhcpServerSuccessTransaction: Option<i32>,
pub dhcpServerLatency: Option<f64>,
pub dhcpServerDOLatency: Option<f64>,
pub dhcpServerRALatency: Option<f64>,
pub maxRoamingDuration: Option<String>,
pub upnOwner: Option<String>,
pub connectedUpn: Option<String>,
pub connectedUpnOwner: Option<String>,
pub connectedUpnId: Option<String>,
pub isGuestUPNEndpoint: Option<bool>,
pub wlcName: Option<String>,
pub wlcUuid: Option<String>,
pub sessionDuration: Option<String>,
pub intelCapable: Option<bool>,
pub hwModel: Option<String>,
pub powerType: Option<String>,
```

```
    pub modelName: Option<String>,
    pub bridgeVMMode: Option<String>,
    pub dhcpNakIp: Option<String>,
    pub dhcpDeclineIp: Option<String>,
    pub portDescription: Option<String>,
    pub latencyVoice: Option<f64>,
    pub latencyVideo: Option<f64>,
    pub latencyBg: Option<f64>,
    pub latencyBe: Option<f64>,
    pub trustScore: Option<String>,
    pub trustDetails: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct HealthScore {
    pub healthType: Option<String>,
    pub reason: Option<String>,
    pub score: Option<i32>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ConnectedDevice {
    #[serde(rename = "type")]
    pub device_type: Option<String>,
    pub name: Option<String>,
    pub mac: Option<String>,
    pub id: Option<String>,
    #[serde(rename = "ip address")]
    pub ip_address: Option<String>,
    pub mgmtIp: Option<String>,
    pub band: Option<String>,
    pub mode: Option<String>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct Onboarding {
    pub averageRunDuration: Option<String>,
    pub maxRunDuration: Option<String>,
    pub averageAssocDuration: Option<String>,
    pub maxAssocDuration: Option<String>,
    pub averageAuthDuration: Option<String>,
    pub maxAuthDuration: Option<String>,
    pub averageDhcpDuration: Option<String>,
    pub maxDhcpDuration: Option<String>,
    pub aaaServerIp: Option<String>,
    pub dhcpServerIp: Option<String>,
```

```
    pub authDoneTime: Option<u64>,   // Changed from Option<String> to Opti
    pub assocDoneTime: Option<u64>,  // Changed from Option<String> to Opti
    pub dhcpDoneTime: Option<u64>,   // Changed from Option<String> to Opti
    pub assocRootcauseList: Option<Vec<String>>,
    pub aaaRootcauseList: Option<Vec<String>>,
    pub dhcpRootcauseList: Option<Vec<String>>,
    pub otherRootcauseList: Option<Vec<String>>,
    pub latestRootCauseList: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ConnectionInfo {
    pub hostType: Option<String>,
    pub nwDeviceName: Option<String>,
    pub nwDeviceMac: Option<String>,
    pub protocol: Option<String>,
    pub band: Option<String>,
    pub spatialStream: Option<String>,
    pub channel: Option<String>,
    pub channelWidth: Option<String>,
    pub wmm: Option<String>,
    pub uapsd: Option<String>,
    pub timestamp: Option<u64>, // Changed from Option<String> to Option<u6
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct Topology {
    pub nodes: Option<Vec<TopologyNode>>,
    pub links: Option<Vec<TopologyLink>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyNode {
    pub role: Option<String>,
    pub name: Option<String>,
    pub id: Option<String>,
    pub description: Option<String>,
    pub deviceType: Option<String>,
    pub platformId: Option<String>,
    pub family: Option<String>,
    pub ip: Option<String>,
    pub softwareVersion: Option<String>,
    pub userId: Option<String>,
    pub nodeType: Option<String>,
    pub radioFrequency: Option<String>,
    pub clients: Option<f64>,
```

```
    pub count: Option<i32>,
    pub healthScore: Option<f64>,
    pub level: Option<f64>,
    pub fabricGroup: Option<String>,
    pub connectedDevice: Option<String>,
    pub fabricRole: Option<Vec<String>>,
    pub stackType: Option<String>,
    pub ipv6: Option<Vec<String>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyLink {
    pub source: Option<String>,
    pub linkStatus: Option<String>,
    pub label: Option<Vec<String>>,
    pub target: Option<String>,
    pub id: Option<String>,
    pub portUtilization: Option<f64>,
}

pub async fn get_client_detail(
    config: &Config,
    token: &Token,
    mac_address: &str,
) -> Result<ClientDetailResponse> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/client-detail", config.dnac_url

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .query(&[("macAddress", mac_address)])
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve client details: {}",
            resp.status()
        ));
    }

    let client_detail_response = resp.json::<ClientDetailResponse>().await?
    Ok(client_detail_response)
}
```

File: ./src/api/clients/mod.rs

```
pub mod getclientdetail;
pub mod getclientenrichment;
```

File: ./src/api/mod.rs

```
pub mod authentication;
pub mod devices;
pub mod clients;
pub mod issues;
pub mod wireless;
```

File: ./src/api/wireless/mod.rs

```
// src/api/wireless/mod.rs

pub mod accesspointconfig;
```

File: ./src/api/wireless/accesspointconfig.rs

```
// src/api/wireless/getaccesspointconfig.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct ApConfig {
    pub instanceUuid: Option<serde_json::Value>,
    pub instanceId: Option<f64>,
    pub authEntityId: Option<serde_json::Value>,
    pub displayName: Option<String>,
    pub authEntityClass: Option<serde_json::Value>,
    pub instanceTenantId: Option<String>,
    pub _orderedListOEIndex: Option<f64>,
    pub _orderedListOEAssocName: Option<serde_json::Value>,
    pub _creationOrderIndex: Option<f64>,
    pub _isBeingChanged: Option<bool>,
    pub deployPending: Option<String>,
    pub instanceCreatedOn: Option<serde_json::Value>,
    pub instanceUpdatedOn: Option<serde_json::Value>,
```

```
    pub changeLogList: Option<serde_json::Value>,
    pub instanceOrigin: Option<serde_json::Value>,
    pub lazyLoadedEntities: Option<serde_json::Value>,
    pub instanceVersion: Option<f64>,
    pub adminStatus: Option<String>,
    pub apHeight: Option<f64>,
    pub apMode: Option<String>,
    pub apName: Option<String>,
    pub ethMac: Option<String>,
    pub failoverPriority: Option<String>,
    pub ledBrightnessLevel: Option<i64>,
    pub ledStatus: Option<String>,
    pub location: Option<String>,
    pub macAddress: Option<String>,
    pub primaryControllerName: Option<String>,
    pub primaryIpAddress: Option<String>,
    pub secondaryControllerName: Option<String>,
    pub secondaryIpAddress: Option<String>,
    pub tertiaryControllerName: Option<String>,
    pub tertiaryIpAddress: Option<String>,
    pub meshDTOs: Option<Vec<serde_json::Value>>,
    pub radioDTOs: Option<Vec<RadioDTO>>,
    pub internalKey: Option<InternalKey>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct RadioDTO {
    pub instanceUuid: Option<serde_json::Value>,
    pub instanceId: Option<f64>,
    pub authEntityId: Option<serde_json::Value>,
    pub displayName: Option<String>,
    pub authEntityClass: Option<serde_json::Value>,
    pub instanceTenantId: Option<String>,
    pub _orderedListOEIndex: Option<f64>,
    pub _orderedListOEAssocName: Option<serde_json::Value>,
    pub _creationOrderIndex: Option<f64>,
    pub _isBeingChanged: Option<bool>,
    pub deployPending: Option<String>,
    pub instanceCreatedOn: Option<serde_json::Value>,
    pub instanceUpdatedOn: Option<serde_json::Value>,
    pub changeLogList: Option<serde_json::Value>,
    pub instanceOrigin: Option<serde_json::Value>,
    pub lazyLoadedEntities: Option<serde_json::Value>,
    pub instanceVersion: Option<f64>,
    pub adminStatus: Option<String>,
    pub antennaAngle: Option<f64>,
    pub antennaElevAngle: Option<f64>,
    pub antennaGain: Option<i64>,
    pub antennaPatternName: Option<String>,
    pub channelAssignmentMode: Option<String>,
```

```
    pub channelNumber: Option<i64>,
    pub channelWidth: Option<String>,
    pub cleanAirSI: Option<String>,
    pub ifType: Option<i64>,
    pub ifTypeValue: Option<String>,
    pub macAddress: Option<String>,
    pub powerAssignmentMode: Option<String>,
    pub powerlevel: Option<i64>,
    pub radioBand: Option<serde_json::Value>,
    pub radioRoleAssignment: Option<serde_json::Value>,
    pub slotId: Option<i64>,
    pub internalKey: Option<InternalKey>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct InternalKey {
    #[serde(rename = "type")]
    pub type_field: Option<String>,
    pub id: Option<f64>,
    pub longType: Option<String>,
    pub url: Option<String>,
}

pub async fn get_ap_config(
    config: &Config,
    token: &Token,
    mac_address: &str,
) -> Result<ApConfig> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/wireless/accesspoint-configurat

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .query(&[("key", mac_address)])
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve AP config: {}",
            resp.status()
        ));
    }

    let ap_config = resp.json::<ApConfig>().await?;
    Ok(ap_config)
```

```
}
```

File: ./src/api/issues/getissuelist.rs

```
// src/api/issues/getissuelist.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;
use std::collections::HashMap;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct IssueListResponse {
    pub version: Option<String>,
    pub totalCount: Option<String>,
    pub response: Option<Vec<Issue>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct Issue {
    pub issueId: Option<String>,
    pub name: Option<String>,
    pub siteId: Option<String>,
    pub deviceId: Option<String>,
    pub deviceRole: Option<String>,
    pub aiDriven: Option<String>,
    pub clientMac: Option<String>,
    pub issue_occurence_count: Option<i32>,
    pub status: Option<String>,
    pub priority: Option<String>,
    pub category: Option<String>,
    pub last_occurence_time: Option<i64>,
}

pub async fn get_issue_list(
    config: &Config,
    token: &Token,
    search_params: &HashMap<String, String>,
) -> Result<IssueListResponse> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/issues", config.dnac_url);
```

```
    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .query(&search_params)
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve issue list: {}",
            resp.status()
        ));
    }

    let issue_list_response = resp.json::<IssueListResponse>().await?;
    Ok(issue_list_response)
}
```

File: ./src/api/issues/mod.rs

```
pub mod getissuelist;
```

File: ./src/api/authentication/auth.rs

```
use crate::app::config::Config;
use crate::helpers::utils;
use anyhow::{anyhow, Result};
use argon2::{
    password_hash::{PasswordHasher, PasswordVerifier, SaltString},
    Argon2,
};
use rand::rngs::OsRng;
use rusqlite::{params, Connection};
use std::fs;
use std::path::PathBuf;

use reqwest::Client;
use serde::Deserialize;

#[derive(Deserialize)]
#[allow(non_snake_case)]
struct TokenResponse {
    Token: String,
}

#[derive(Clone)]
```

```rust
#[allow(non_snake_case)]
pub struct Token {
    pub value: String,
    pub obtained_at: u64,
    pub expires_at: u64,
}



pub async fn authenticate(config: &Config) -> Result<Token> {
    // Check for existing token
    if let Some(token) = load_token()? {
        if token.expires_at > utils::current_timestamp() {
            // Token is still valid
            return Ok(token);
        }
    }

    // Token is missing or expired; proceed to authenticate
    let credentials = load_credentials(&config.username)?;
    let password = prompt_password(&config.username)?;
    if !verify_password(&password, &credentials.password_hash)? {
        return Err(anyhow!("Invalid password"));
    }

    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let auth_url = format!("{}/dna/system/api/v1/auth/token", config.dnac_u

    let resp = client
        .post(&auth_url)
        .basic_auth(&config.username, Some(&password))
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Authentication failed with status: {}",
            resp.status()
        ));
    }

    let token_resp: TokenResponse = resp.json().await?;

    let obtained_at = utils::current_timestamp();
    let expires_at = obtained_at + 1 * 60 * 60; // Token valid for 1 hour

    let token = Token {
        value: token_resp.Token,
        obtained_at,
```

```
        expires_at,
    };

    // Store the token
    store_token(&token)?;

    Ok(token)
}

struct StoredCredentials {
    password_hash: String,
}

fn get_db_path() -> PathBuf {
    let mut db_path = dirs::config_dir().unwrap();
    db_path.push("catsh");
    db_path.push("credentials.db");
    db_path
}

fn load_credentials(username: &str) -> Result<StoredCredentials> {
    let db_path = get_db_path();
    if !db_path.exists() {
        println!("No credentials database found. Starting setup...");
        store_credentials(username)?;
    }

    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    let mut stmt = conn.prepare("SELECT password_hash FROM credentials WHER
    let mut rows = stmt.query(params![username])?;

    if let Some(row) = rows.next()? {
        let password_hash: String = row.get(0)?;
        Ok(StoredCredentials { password_hash })
    } else {
        println!("No credentials found for user '{}'. Starting setup...", u
        store_credentials(username)?;

        // After storing credentials, try to load them again
        let mut stmt = conn.prepare("SELECT password_hash FROM credentials
        let mut rows = stmt.query(params![username])?;

        if let Some(row) = rows.next()? {
            let password_hash: String = row.get(0)?;
            Ok(StoredCredentials { password_hash })
        } else {
            Err(anyhow!("Credentials not found after setup"))
        }
    }
```

```
}

fn store_credentials(username: &str) -> Result<()> {
    let password = prompt_new_password(username)?;

    let db_path = get_db_path();
    fs::create_dir_all(db_path.parent().unwrap())?;
    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    let salt = SaltString::generate(&mut OsRng);
    let argon2 = Argon2::default();
    let password_hash = argon2
        .hash_password(password.as_bytes(), &salt)
        .map_err(|e| anyhow!(e))?
        .to_string();

    conn.execute(
        "INSERT INTO credentials (username, password_hash) VALUES (?1, ?2)"
        params![username, password_hash],
    )?;

    Ok(())
}

fn prompt_password(username: &str) -> Result<String> {
    let password = rpassword::prompt_password(format!("Enter password for {
    Ok(password)
}

fn prompt_new_password(username: &str) -> Result<String> {
    let password = rpassword::prompt_password(format!("Set a new password f
    let confirm_password = rpassword::prompt_password("Confirm password: ")

    if password != confirm_password {
        return Err(anyhow!("Passwords do not match"));
    }

    Ok(password)
}

fn verify_password(password: &str, password_hash: &str) -> Result<bool> {
    let parsed_hash = argon2::password_hash::PasswordHash::new(password_has
        .map_err(|e| anyhow!(e))?;
    let argon2 = Argon2::default();
    Ok(argon2
        .verify_password(password.as_bytes(), &parsed_hash)
        .is_ok())
}

// Functions to store and load the token
```

```
fn store_token(token: &Token) -> Result<()> {
    let db_path = get_db_path();
    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    conn.execute(
        "DELETE FROM token", // Clear any existing token
        [],
    )?;

    conn.execute(
        "INSERT INTO token (value, obtained_at, expires_at) VALUES (?1, ?2,
        params![token.value, token.obtained_at, token.expires_at],
    )?;

    Ok(())
}

fn load_token() -> Result<Option<Token>> {
    let db_path = get_db_path();
    let conn = Connection::open(db_path)?;

    create_tables(&conn)?;

    let mut stmt = conn.prepare("SELECT value, obtained_at, expires_at FROM
    let mut rows = stmt.query([])?;

    if let Some(row) = rows.next()? {
        let value: String = row.get(0)?;
        let obtained_at: u64 = row.get(1)?;
        let expires_at: u64 = row.get(2)?;

        Ok(Some(Token {
            value,
            obtained_at,
            expires_at,
        }))
    } else {
        Ok(None)
    }
}

fn create_tables(conn: &Connection) -> Result<()> {
    conn.execute(
        "CREATE TABLE IF NOT EXISTS credentials (
            id INTEGER PRIMARY KEY,
            username TEXT NOT NULL,
            password_hash TEXT NOT NULL
        )",
        [],
    )?;
```

```
    conn.execute(
        "CREATE TABLE IF NOT EXISTS token (
            id INTEGER PRIMARY KEY,
            value TEXT NOT NULL,
            obtained_at INTEGER NOT NULL,
            expires_at INTEGER NOT NULL
        )",
        [],
    )?;

    Ok(())
}
```

File: ./src/api/authentication/mod.rs

```
pub mod auth;
```

File: ./src/api/devices/getdevicelist.rs

```
use crate::api::authentication::auth::{self, Token};
use crate::app::config::Config;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[serde(rename_all = "camelCase")]
#[allow(dead_code)]
pub struct AllDevices {
    pub reachability_failure_reason: Option<String>,
    pub reachability_status: Option<String>,
    pub series: Option<String>,
    pub snmp_contact: Option<String>,
    pub snmp_location: Option<String>,
    pub tag_count: Option<String>,
    pub tunnel_udp_port: Option<serde_json::Value>, // Use `serde_json::Val
    pub uptime_seconds: Option<i64>, // Assuming "integer" corresponds to i
    pub waas_device_mode: Option<serde_json::Value>,
    pub serial_number: Option<String>,
    pub last_update_time: Option<i64>,
    pub mac_address: Option<String>,
    pub up_time: Option<String>,
    pub device_support_level: Option<String>,
    pub hostname: Option<String>,
    pub device_type: Option<String>, // Renamed "type" to "device_type" to
```

```
    pub memory_size: Option<String>,
    pub family: Option<String>,
    pub error_code: Option<String>,
    pub software_type: Option<String>,
    pub software_version: Option<String>,
    pub description: Option<String>,
    pub role_source: Option<String>,
    pub location: Option<serde_json::Value>,
    pub role: Option<String>,
    pub collection_interval: Option<String>,
    pub inventory_status_detail: Option<String>,
    pub ap_ethernet_mac_address: Option<String>,
    pub ap_manager_interface_ip: Option<String>,
    pub associated_wlc_ip: Option<String>,
    pub boot_date_time: Option<String>,
    pub collection_status: Option<String>,
    pub error_description: Option<String>,
    pub interface_count: Option<String>,
    pub last_updated: Option<String>,
    pub line_card_count: Option<String>,
    pub line_card_id: Option<String>,
    pub location_name: Option<serde_json::Value>,
    pub managed_atleast_once: Option<bool>,
    pub management_ip_address: Option<String>,
    pub platform_id: Option<String>,
    pub management_state: Option<String>,
    pub instance_tenant_id: Option<String>,
    pub instance_uuid: Option<String>,
    pub id: Option<String>,
}

#[derive(Debug, Deserialize)]
struct DevicesResponse {
    response: Vec<AllDevices>,
}

pub async fn get_all_devices(config: &Config, token: &Token) -> Result<Vec<
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let mut all_devices: Vec<AllDevices> = Vec::new();
    let mut offset = 1; // Adjust based on API documentation (could be 0)
    let limit = 500;    // Set the limit as per API maximum

    loop {
        let devices_url = format!(
            "{}/dna/intent/api/v1/network-device?offset={}&limit={}",
            config.dnac_url, offset, limit
        );

        // Perform the API request with reauthentication handling
```

```
        let devices_response: DevicesResponse =
            send_authenticated_request(&client, config, token, &devices_url

        let devices = devices_response.response;

        if devices.is_empty() {
            // No more devices to fetch
            break;
        }

        all_devices.extend(devices);

        // Increment offset
        offset += limit;
    }

    Ok(all_devices)
}

/// Sends an authenticated GET request, handling reauthentication if necess
async fn send_authenticated_request<T: serde::de::DeserializeOwned>(
    client: &Client,
    config: &Config,
    token: &Token,
    url: &str,
) -> Result<T> {
    let mut current_token = token.clone();

    loop {
        let resp = client
            .get(url)
            .header("X-Auth-Token", &current_token.value)
            .send()
            .await?;

        if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
            // Token expired or invalid, reauthenticate and retry
            eprintln!("Token expired. Reauthenticating...");
            current_token = auth::authenticate(config).await?;
            continue; // Retry the request with the new token
        }

        if !resp.status().is_success() {
            return Err(anyhow!(
                "Failed to complete request: {}",
                resp.status()
            ));
        }

        // Deserialize and return the response
        let result = resp.json::<T>().await?;
        return Ok(result);
```

```
    }
}
```

File: ./src/api/devices/devicedetailenrichment.rs

```
// src/api/devices/devicedetailenrichment.rs

use crate::app::config::Config;
use crate::api::authentication::auth::Token;
use anyhow::{anyhow, Result};
use reqwest::Client;
use serde::Deserialize;

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct DeviceEnrichmentResponse {
    pub deviceDetails: DeviceDetails,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct DeviceDetails {
    pub family: Option<String>,
    #[serde(rename = "type")]
    pub type_field: Option<String>,
    pub location: Option<serde_json::Value>,
    pub errorCode: Option<String>,
    pub macAddress: Option<String>,
    pub role: Option<String>,
    pub apManagerInterfaceIp: Option<String>,
    pub associatedWlcIp: Option<String>,
    pub bootDateTime: Option<String>,
    pub collectionStatus: Option<String>,
    pub interfaceCount: Option<String>,
    pub lineCardCount: Option<String>,
    pub lineCardId: Option<String>,
    pub managementIpAddress: Option<String>,
    pub memorySize: Option<String>,
    pub platformId: Option<String>,
    pub reachabilityFailureReason: Option<String>,
    pub reachabilityStatus: Option<String>,
    pub snmpContact: Option<String>,
    pub snmpLocation: Option<String>,
    pub tunnelUdpPort: Option<serde_json::Value>,
    pub waasDeviceMode: Option<serde_json::Value>,
    pub series: Option<String>,
    pub inventoryStatusDetail: Option<String>,
    pub collectionInterval: Option<String>,
```

```
    pub serialNumber: Option<String>,
    pub softwareVersion: Option<String>,
    pub roleSource: Option<String>,
    pub hostname: Option<String>,
    pub upTime: Option<String>,
    pub lastUpdateTime: Option<i64>,
    pub errorDescription: Option<String>,
    pub locationName: Option<serde_json::Value>,
    pub tagCount: Option<String>,
    pub lastUpdated: Option<String>,
    pub instanceUuid: Option<String>,
    pub id: Option<String>,
    pub neighborTopology: Option<Vec<NeighborTopology>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct NeighborTopology {
    pub nodes: Option<Vec<TopologyNode>>,
    pub links: Option<Vec<TopologyLink>>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyNode {
    pub role: Option<String>,
    pub name: Option<String>,
    pub id: Option<String>,
    pub description: Option<String>,
    pub deviceType: Option<String>,
    pub platformId: Option<String>,
    pub family: Option<String>,
    pub ip: Option<String>,
    pub softwareVersion: Option<String>,
    pub userId: Option<serde_json::Value>,
    pub nodeType: Option<String>,
    pub radioFrequency: Option<serde_json::Value>,
    pub clients: Option<serde_json::Value>,
    pub count: Option<serde_json::Value>,
    pub healthScore: Option<i32>,
    pub level: Option<f64>,
    pub fabricGroup: Option<serde_json::Value>,
    pub connectedDevice: Option<serde_json::Value>,
}

#[derive(Debug, Deserialize)]
#[allow(non_snake_case)]
#[allow(dead_code)]
pub struct TopologyLink {
    pub source: Option<String>,
```

```
    pub linkStatus: Option<String>,
    pub label: Option<Vec<String>>,
    pub target: Option<String>,
    pub id: Option<serde_json::Value>,
    pub portUtilization: Option<serde_json::Value>,
}

pub async fn get_device_enrichment(
    config: &Config,
    token: &Token,
    entity_type: &str,
    entity_value: &str,
) -> Result<DeviceDetails> {
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    let url = format!("{}/dna/intent/api/v1/device-enrichment-details", con

    let resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .header("entity_type", entity_type)
        .header("entity_value", entity_value)
        .send()
        .await?;

    if !resp.status().is_success() {
        return Err(anyhow!(
            "Failed to retrieve device enrichment details: {}",
            resp.status()
        ));
    }

    let enrichment_responses = resp.json::<Vec<DeviceEnrichmentResponse>>()

    if let Some(first_response) = enrichment_responses.into_iter().next() {
        Ok(first_response.deviceDetails)
    } else {
        Err(anyhow!("No device enrichment details found"))
    }
}
```

File: ./src/api/devices/mod.rs

```
// src/api/devices/mod.rs

pub mod getdevicelist;
pub mod devicedetailenrichment;
```

File: ./src/templates/exampleintegration.rs

```
fn handle_example(command: ExampleSubcommands) {
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        let token = match api::auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
                error!("Authentication failed: {}", e);
                return;
            }
        };

        if let Err(e) = handle_example_command(config, token, command).awai
            error!("Error handling example command: {}", e);
        }
    });
}
```

File: ./src/templates/templates.rs

```
use clap::{Args, Parser, Subcommand};
use reqwest::{Client, Method};
use serde::{Deserialize, Serialize};
use anyhow::{Result, anyhow};
use crate::config::Config;
use crate::api::auth::Token;

// Root command template
#[derive(Debug, Parser)]
#[command(name = "catsh", about = "A REPL CLI for managing network devices"
pub struct Cli {
    #[command(subcommand)]
    pub command: Commands,
}

// Main command structure
#[derive(Debug, Subcommand)]
pub enum Commands {
    ExampleCommand {
        #[command(subcommand)]
        subcommand: ExampleSubcommands,
```

```
    },
}

// Subcommands template
#[derive(Debug, Subcommand)]
pub enum ExampleSubcommands {
    List,
    Details {
        #[arg(help = "ID of the item to retrieve details for")]
        id: String,
    },
    Create(NewItemArgs),
}

// Arguments for a subcommand
#[derive(Debug, Args)]
pub struct NewItemArgs {
    #[arg(help = "Name of the new item")]
    pub name: String,
    #[arg(help = "Description of the new item")]
    pub description: String,
}

// Handle ExampleCommand
pub async fn handle_example_command(config: Config, token: Token, subcomman
    let client = Client::builder()
        .danger_accept_invalid_certs(!config.verify_ssl)
        .build()?;

    match subcommand {
        ExampleSubcommands::List => {
            let response = api_get::<Vec<Item>>(&client, &config, &token, "
            println!("Items: {:?}", response);
        }
        ExampleSubcommands::Details { id } => {
            let response = api_get::<Item>(&client, &config, &token, &forma
            println!("Item Details: {:?}", response);
        }
        ExampleSubcommands::Create(args) => {
            let new_item = NewItem {
                name: args.name,
                description: args.description,
            };
            let response = api_post::<Item, NewItem>(&client, &config, &tok
            println!("Created Item: {:?}", response);
        }
    }

    Ok(())
}

// Example Data Structures
```

```rust
#[derive(Debug, Deserialize)]
pub struct Item {
    pub id: String,
    pub name: String,
    pub description: String,
}

#[derive(Debug, Serialize)]
pub struct NewItem {
    pub name: String,
    pub description: String,
}

// API Call Templates
pub async fn api_get<T: Deserialize<'static>>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .get(&url)
        .header("X-Auth-Token", &token.value)
        .send()
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("GET request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}

pub async fn api_post<T: Deserialize<'static>, U: Serialize>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
    body: &U,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .post(&url)
        .header("X-Auth-Token", &token.value)
        .json(body)
        .send()
```

```
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("POST request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}

pub async fn api_put<T: Deserialize<'static>, U: Serialize>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
    body: &U,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .put(&url)
        .header("X-Auth-Token", &token.value)
        .json(body)
        .send()
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("PUT request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}

pub async fn api_delete<T: Deserialize<'static>>(
    client: &Client,
    config: &Config,
    token: &Token,
    endpoint: &str,
) -> Result<T> {
    let url = format!("{}/{}", config.dnac_url, endpoint);
    let mut resp = client
        .delete(&url)
        .header("X-Auth-Token", &token.value)
        .send()
```

```
        .await?;

    if resp.status() == reqwest::StatusCode::UNAUTHORIZED {
        return Err(anyhow!("Unauthorized: Token may have expired"));
    }

    if !resp.status().is_success() {
        return Err(anyhow!("DELETE request failed: {}", resp.status()));
    }

    let result = resp.json::<T>().await?;
    Ok(result)
}
```

File: ./src/commands/app/config.rs

```
use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum AppConfigCommands {
    /// Reset app configuration
    Reset,
}
```

File: ./src/commands/app/update.rs

```
// Empty file since `Update` doesn't have subcommands
```

File: ./src/commands/app/mod.rs

```
pub mod config;
pub mod update; // Added this line

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum AppCommands {
    /// App configuration commands
    Config {
        #[command(subcommand)]
        subcommand: config::AppConfigCommands,
    },
    /// Update the program to the latest release available (Program restart
    Update,
```

```
    // Additional app-related subcommands can be added here
}
```

File: ./src/commands/config/commands.rs

File: ./src/commands/config/mod.rs

```
// This module is currently empty since the config command starts a sub-REP
```

File: ./src/commands/show/ap.rs

```
// src/commands/show/ap.rs

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum ApCommands {
    /// Show AP configuration by MAC address
    Config {
        /// MAC address of the AP
        mac_address: String,
    },
}
```

File: ./src/commands/show/device.rs

```
// src/commands/show/device.rs

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum DeviceCommands {
    /// List devices
    List {
        #[command(subcommand)]
        filter: DeviceListFilter,
    },
    /// Show device details
    Detail {
        #[command(subcommand)]
        filter: DeviceDetailFilter,
```

```
    },
    /// Show device enrichment detail
    Enrichment {
        #[command(subcommand)]
        filter: DeviceEnrichmentFilter,
    },
}

#[derive(Debug, Subcommand)]
pub enum DeviceListFilter {
    /// List all devices
    All,
    /// List devices filtered by hostname
    Hostname {
        /// Optional partial hostname to filter by
        partial_hostname: Option<String>,
    },
    /// List devices filtered by IP address
    Ip {
        /// Optional partial IP address to filter by
        partial_ip: Option<String>,
    },
    /// List devices filtered by WLC IP address
    Wlc {
        /// Optional partial WLC ip to filter by
        partial_wlc: Option<String>,
    },
}

#[derive(Debug, Subcommand)]
pub enum DeviceDetailFilter {
    /// Show device detail by hostname
    Hostname {
        /// The hostname of the device
        hostname: String,
    },
    /// Show device detail by MAC address
    Mac {
        /// The MAC address of the device
        mac_address: String,
    },
    /// Show device detail by IP address
    Ip {
        /// The IP address of the device
        ip_address: String,
    },
}

#[derive(Debug, Subcommand)]
pub enum DeviceEnrichmentFilter {
    /// Enrichment by MAC address
    Mac {
```

```
        /// The MAC address of the device
        mac_address: String,
    },
    /// Enrichment by IP address
    Ip {
        /// The IP address of the device
        ip_address: String,
    },
}
```

File: ./src/commands/show/client.rs

```
// src/commands/show/client.rs

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum ClientCommands {
    /// Show client details by MAC address
    Detail {
        /// MAC address of the client
        mac_address: String,
    },
    /// Show client enrichment by network user ID or MAC address
    Enrichment {
        /// The entity type (network_user_id or mac_address)
        #[arg(value_parser = ["network_user_id", "mac_address"])]
        entity_type: String,
        /// The value of the entity (user ID or MAC address)
        entity_value: String,
        /// Optional issue category
        #[arg(long)]
        issue_category: Option<String>,
    },
}
```

File: ./src/commands/show/mod.rs

```
pub mod device;
pub mod client;
pub mod issue;
pub mod ap;

use clap::Subcommand;

#[derive(Debug, Subcommand)]
pub enum ShowCommands {
    /// Show device information
```

```
    Device {
        #[command(subcommand)]
        subcommand: device::DeviceCommands,
    },
    /// Show client information
    Client {
        #[command(subcommand)]
        subcommand: client::ClientCommands,
    },
    /// Show issues in Catalyst Center
    Issue {
        #[command(subcommand)]
        subcommand: issue::IssueCommands,
    },
    /// Show Access Point information
    Ap {
        #[command(subcommand)]
        subcommand: ap::ApCommands,
    },
}
```

File: ./src/commands/show/issue.rs

```
// src/commands/show/issue.rs

#[allow(unused_imports)]
use clap::{Parser, Subcommand, ValueEnum};

#[derive(Debug, Subcommand)]
#[allow(unused_imports)]
pub enum IssueCommands {
    /// List issues based on search criteria
    List {
        /// Search option (e.g., deviceId, macAddress, priority, etc.)
        #[arg(value_enum)]
        search_option: Option<SearchOption>,
        /// Search input corresponding to the search option
        search_input: Option<String>,
    },
}

#[derive(Debug, Clone, ValueEnum)]
#[allow(dead_code)]
pub enum SearchOption {
    /// Start time to search from when looking for issues
    StartTime,
    /// End time used in conjunction with StartTime
    EndTime,
    /// SiteID gotten from a show site detail command
```

```
    SiteId,
    /// DeviceID gotten from a show device detail command
    DeviceId,
    /// MAC Address of a device or client
    MacAddress,
    /// One of these options - P1, P2, P3, P4
    Priority,
    /// Only pull issues are/aren't AI Driven - must be "Yes" or "No"
    AiDriven,
    /// Only pull issues with a specific status
    IssueStatus,
}
```

File: ./src/commands/mod.rs

```
pub mod show;
pub mod config;
pub mod app;

use clap::{Parser, Subcommand};
use crate::handlers::{handle_show_command, handle_config_command, handle_ap

#[derive(Debug, Parser)]
#[command(name = "catsh", about = "A command line interface for Cisco Catal
pub struct Cli {
    #[command(subcommand)]
    pub command: Commands,
}

#[derive(Debug, Subcommand)]
pub enum Commands {
    /// Show commands
    Show {
        #[command(subcommand)]
        subcommand: show::ShowCommands,
    },
    /// Start configuration sub-REPL
    Config,
    /// App-specific commands
    App {
        #[command(subcommand)]
        subcommand: app::AppCommands,
    },
    /// Exit the program
    Exit,
}

pub fn route_command(command: Commands) {
    match command {
        Commands::Show { subcommand } => handle_show_command(subcommand),
```

```
        Commands::Config => handle_config_command(),
        Commands::App { subcommand } => handle_app_command(subcommand),
        Commands::Exit => {
            println!("Exiting catsh...");
            std::process::exit(0);
        }
    }
}
```

File: ./src/handlers/app/config.rs

```
use log::error;
use crate::app::config; // Adjusted import
use crate::commands::app::config::AppConfigCommands;

pub fn handle_app_config_command(subcommand: AppConfigCommands) {
    match subcommand {
        AppConfigCommands::Reset => {
            if let Err(e) = config::reset_config() {
                error!("Failed to reset configuration: {}", e);
            } else {
                println!("Configuration reset successfully.");
            }
        }
    }
}
```

File: ./src/handlers/app/update.rs

```
#[allow(unused_imports)]
use crate::app::update; // Corrected import

pub fn handle_update_command() {
    #[cfg(any(target_os = "linux", target_os = "macos"))]
    {
        if let Err(e) = update::update_to_latest() {
            eprintln!("Update failed: {}", e);
        } else {
            println!("Update completed successfully.");
        }
    }

    #[cfg(target_os = "windows")]
    {
        println!("Please download and run the latest `windows_installer.exe
    }
```

```
}
```

File: ./src/handlers/app/mod.rs

```
pub mod config;
pub mod update;

use crate::commands::app::AppCommands;

pub fn handle_app_command(subcommand: AppCommands) {
    match subcommand {
        AppCommands::Config { subcommand } => config::handle_app_config_com
        AppCommands::Update => update::handle_update_command(),
        // Handle other app subcommands here
    }
}
```

File: ./src/handlers/config/repl.rs

```
use clap_repl::reedline::{DefaultPrompt, DefaultPromptSegment, FileBackedHi
use clap_repl::ClapEditor;
use clap::{Parser, Subcommand};

#[derive(Debug, Parser)]
#[command(name = "", about = "Configuration Mode REPL")]
struct ConfigCli {
    #[command(subcommand)]
    command: ConfigCommands,
}

#[derive(Debug, Subcommand)]
enum ConfigCommands {
    /// Dummy command for demonstration
    Dummy,
    /// Exit configuration mode
    Exit,
    /// End configuration mode
    End,
}

pub fn start_config_repl() {
    println!("Entering configuration mode. Type 'exit' or 'end' to leave.")
    let prompt = DefaultPrompt {
        left_prompt: DefaultPromptSegment::Basic("catsh(config)#".to_owned(
        ..DefaultPrompt::default()
    };
```

```
    let rl = ClapEditor::<ConfigCli>::builder()
        .with_prompt(Box::new(prompt))
        .with_editor_hook(|reed| {
            reed.with_history(Box::new(
                FileBackedHistory::with_file(10000, "/tmp/catsh-config-cli-
            ))
        })
        .build();

    rl.repl(|cli| {
        match cli.command {
            ConfigCommands::Dummy => {
                println!("Dummy command executed in config mode.");
            }
            ConfigCommands::Exit | ConfigCommands::End => {
                println!("Exiting configuration mode.");
                std::process::exit(0);
            }
        }
    });
}
```

File: ./src/handlers/config/mod.rs

```
pub mod repl;

pub fn handle_config_command() {
    repl::start_config_repl();
}
```

File: ./src/handlers/show/ap.rs

```
// src/handlers/show/ap.rs

use crate::commands::show::ap::ApCommands;
use crate::app::config;
use crate::api::authentication::auth;
use crate::api::wireless::accesspointconfig;
use crate::helpers::utils;
use log::error;

pub fn handle_ap_command(subcommand: ApCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
```

```
        // Load configuration
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        // Authenticate and get token
        let token = match auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
                error!("Authentication failed: {}", e);
                return;
            }
        };

        match subcommand {
            ApCommands::Config { mac_address } => {
                // Fetch AP config
                match accesspointconfig::get_ap_config(&config, &token, &ma
                    Ok(ap_config) => {
                        utils::print_ap_config(ap_config);
                    }
                    Err(e) => {
                        error!("Failed to retrieve AP config: {}", e);
                    }
                }
            }
        }
    });
}
```

File: ./src/handlers/show/device.rs

```
// src/handlers/show/device.rs

use log::error;
use crate::app::config;
use crate::helpers::utils;
use crate::api::authentication::auth;
use crate::api::devices::{devicedetailenrichment, getdevicelist};
use crate::commands::show::device::{
    DeviceCommands, DeviceDetailFilter, DeviceEnrichmentFilter, DeviceListF
};

pub fn handle_device_command(subcommand: DeviceCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
```

```
runtime.block_on(async {
    let config = match config::load_config() {
        Ok(cfg) => cfg,
        Err(e) => {
            error!("Failed to load configuration: {}", e);
            return;
        }
    };

    let token = match auth::authenticate(&config).await {
        Ok(t) => t,
        Err(e) => {
            error!("Authentication failed: {}", e);
            return;
        }
    };

    match subcommand {
        DeviceCommands::List { filter } => {
            // Fetch all devices
            match getdevicelist::get_all_devices(&config, &token).await
                Ok(devices) => {
                    // Apply filter if necessary
                    let filtered_devices = match filter {
                        DeviceListFilter::All => devices,
                        DeviceListFilter::Hostname { partial_hostname }
                            devices
                                .into_iter()
                                .filter(|device| {
                                    if let Some(ref name) = device.host
                                        if let Some(ref partial) = part
                                            name.contains(partial)
                                        } else {
                                            true // Include all devices
                                        }
                                    } else {
                                        false
                                    }
                                })
                                .collect()
                        }
                        DeviceListFilter::Ip { partial_ip } => {
                            devices
                                .into_iter()
                                .filter(|device| {
                                    if let Some(ref ip) = device.manage
                                        if let Some(ref partial) = part
                                            ip.contains(partial)
                                        } else {
                                            true // Include all devices
                                        }
                                    } else {
```

```
                                false
                            }
                        })
                        .collect()
                }
                DeviceListFilter::Wlc { partial_wlc } => {
                    devices
                        .into_iter()
                        .filter(|device| {
                            if let Some(ref wlc_ip) = device.as
                                if let Some(ref partial) = part
                                    wlc_ip.contains(partial)
                                } else {
                                    true // Include all devices
                                }
                            } else {
                                false
                            }
                        })
                        .collect()
                }
            };

            utils::print_devices(filtered_devices);
        }
        Err(e) => error!("Failed to retrieve devices: {}", e),
    }
}
DeviceCommands::Detail { filter } => {
    // Fetch all devices
    match getdevicelist::get_all_devices(&config, &token).await
        Ok(devices) => {
            // Find the device matching the filter
            let device_option = match filter {
                DeviceDetailFilter::Hostname { ref hostname } =
                    .into_iter()
                    .find(|device| device.hostname.as_deref() =
                DeviceDetailFilter::Mac { ref mac_address } =>
                    .into_iter()
                    .find(|device| device.mac_address.as_deref(
                DeviceDetailFilter::Ip { ref ip_address } => de
                    .into_iter()
                    .find(|device| {
                        device.management_ip_address.as_deref()
                    }),
            };

            match device_option {
                Some(device) => utils::print_device_detail(devi
                None => println!("No device found matching the
            }
        }
```

```
                    Err(e) => error!("Failed to retrieve devices: {}", e),
                }
            }
            DeviceCommands::Enrichment { filter } => {
                // Handle the Enrichment command
                match filter {
                    DeviceEnrichmentFilter::Mac { mac_address } => {
                        match devicedetailenrichment::get_device_enrichment
                            &config,
                            &token,
                            "mac_address",
                            &mac_address,
                        )
                        .await
                        {
                            Ok(device_details) => {
                                utils::print_device_enrichment(device_detai
                            }
                            Err(e) => error!("Failed to retrieve device enr
                        }
                    }
                    DeviceEnrichmentFilter::Ip { ip_address } => {
                        match devicedetailenrichment::get_device_enrichment
                            &config,
                            &token,
                            "ip_address",
                            &ip_address,
                        )
                        .await
                        {
                            Ok(device_details) => {
                                utils::print_device_enrichment(device_detai
                            }
                            Err(e) => error!("Failed to retrieve device enr
                        }
                    }
                }
            }
        }
    });
}
```

File: ./src/handlers/show/client.rs

```
// src/handlers/show/client.rs

use crate::commands::show::client::ClientCommands;
use crate::app::config;
use crate::api::authentication::auth;
use crate::api::clients::{getclientdetail, getclientenrichment};
```

```
use crate::helpers::utils;
use log::error;

pub fn handle_client_command(subcommand: ClientCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
    runtime.block_on(async {
        // Load configuration
        let config = match config::load_config() {
            Ok(cfg) => cfg,
            Err(e) => {
                error!("Failed to load configuration: {}", e);
                return;
            }
        };

        // Authenticate and get token
        let token = match auth::authenticate(&config).await {
            Ok(t) => t,
            Err(e) => {
                error!("Authentication failed: {}", e);
                return;
            }
        };

        match subcommand {
            ClientCommands::Detail { mac_address } => {
                // Fetch client details
                match getclientdetail::get_client_detail(&config, &token, &
                    Ok(client_detail_response) => {
                        utils::print_client_detail(client_detail_response);
                    }
                    Err(e) => {
                        error!("Failed to retrieve client details: {}", e);
                    }
                }
            }
            ClientCommands::Enrichment {
                entity_type,
                entity_value,
                issue_category,
            } => {
                // Fetch client enrichment details
                match getclientenrichment::get_client_enrichment(
                    &config,
                    &token,
                    &entity_type,
                    &entity_value,
                    issue_category.as_deref(),
                )
                .await
                {
```

```
                    Ok(enrichment_response) => {
                        utils::print_client_enrichment(enrichment_response)
                    }
                    Err(e) => {
                        error!("Failed to retrieve client enrichment detail
                    }
                }
            }
        }
    });
}
```

File: ./src/handlers/show/mod.rs

```
pub mod device;
pub mod client;
pub mod issue;
pub mod ap;
use crate::commands::show::ShowCommands;

pub fn handle_show_command(subcommand: ShowCommands) {
    match subcommand {
        ShowCommands::Device { subcommand } => device::handle_device_comman

        ShowCommands::Client { subcommand } => client::handle_client_comman

        ShowCommands::Issue { subcommand } => issue::handle_issue_command(s

        ShowCommands::Ap { subcommand } => ap::handle_ap_command(subcommand
    }
}
```

File: ./src/handlers/show/issue.rs

```
// src/handlers/show/issue.rs

use crate::commands::show::issue::{IssueCommands, SearchOption};
use crate::app::config;
use crate::api::authentication::auth;
use crate::api::issues::getissuelist;
use crate::helpers::utils;
use log::error;
use std::collections::HashMap;

pub fn handle_issue_command(subcommand: IssueCommands) {
    // Create a Tokio runtime
    let runtime = tokio::runtime::Runtime::new().expect("Failed to create T
```

```
runtime.block_on(async {
    // Load configuration
    let config = match config::load_config() {
        Ok(cfg) => cfg,
        Err(e) => {
            error!("Failed to load configuration: {}", e);
            return;
        }
    };

    // Authenticate and get token
    let token = match auth::authenticate(&config).await {
        Ok(t) => t,
        Err(e) => {
            error!("Authentication failed: {}", e);
            return;
        }
    };

    match subcommand {
        IssueCommands::List { search_option, search_input } => {
            // Prepare search parameters
            let mut search_params = HashMap::new();

            if let Some(option) = search_option {
                if let Some(input) = search_input {
                    match option {
                        SearchOption::StartTime => {
                            search_params.insert("startTime".to_string(
                        }
                        SearchOption::EndTime => {
                            search_params.insert("endTime".to_string(),
                        }
                        SearchOption::SiteId => {
                            search_params.insert("siteId".to_string(),
                        }
                        SearchOption::DeviceId => {
                            search_params.insert("deviceId".to_string()
                        }
                        SearchOption::MacAddress => {
                            search_params.insert("macAddress".to_string
                        }
                        SearchOption::Priority => {
                            search_params.insert("priority".to_string()
                        }
                        SearchOption::AiDriven => {
                            search_params.insert("aiDriven".to_string()
                        }
                        SearchOption::IssueStatus => {
                            search_params.insert("issueStatus".to_strin
                        }
                    }
```

```
                    } else {
                        error!("Search input is required when a search opti
                        return;
                    }
                }

                // Fetch issue list
                match getissuelist::get_issue_list(&config, &token, &search
                    Ok(issue_list_response) => {
                        utils::print_issue_list(issue_list_response);
                    }
                    Err(e) => {
                        error!("Failed to retrieve issue list: {}", e);
                    }
                }
            }
        }
    });
}
```

File: ./src/handlers/mod.rs

```
pub mod show;
pub mod config;
pub mod app;

pub use show::handle_show_command;
pub use config::handle_config_command;
pub use app::handle_app_command;
```

File: ./src/helpers/mod.rs

```
pub mod utils;
```

File: ./src/helpers/utils.rs

```
// src/helpers/utils.rs
#[allow(unused_imports)]
use crate::api::clients::getclientdetail::{
    ClientDetailResponse,
    HealthScore as ClientDetailHealthScore,
    ConnectedDevice as ClientDetailConnectedDevice,
    Onboarding,
    ConnectionInfo,
```

```
    Topology,
    TopologyNode as ClientDetailTopologyNode,
    TopologyLink as ClientDetailTopologyLink,
};
#[allow(unused_imports)]
use crate::api::clients::getclientenrichment::{
    ClientEnrichmentResponse,
    ClientEnrichment,
    UserDetails,
    HealthScore as ClientEnrichmentHealthScore,
    ConnectedDevice as ClientEnrichmentConnectedDevice,
    DeviceDetails as ClientEnrichmentDeviceDetails,
    NeighborTopology,
    TopologyNode as ClientEnrichmentTopologyNode,
    TopologyLink as ClientEnrichmentTopologyLink,
    IssueDetails,
    Issue as ClientEnrichmentIssue,
    SuggestedAction,
    ImpactedHost,
    ImpactedHostLocation,
    VlanId,
};

#[allow(unused_imports)]
use crate::api::devices::devicedetailenrichment::DeviceDetails as DeviceDet
use crate::api::devices::getdevicelist::AllDevices;
use crate::api::clients::getclientenrichment::StringOrNumber;

#[allow(unused_imports)]
use crate::api::issues::getissuelist::{IssueListResponse, Issue as IssueLis
use crate::api::devices::devicedetailenrichment::DeviceDetails;
use crate::api::wireless::accesspointconfig::ApConfig;

use chrono::{DateTime, Utc};
use prettytable::{row, Table};

pub fn current_timestamp() -> u64 {
    Utc::now().timestamp_millis() as u64
}

// Function to print a list of devices
pub fn print_devices(devices: Vec<AllDevices>) {
    let mut table = Table::new();
    table.add_row(row![
        "Hostname",
        "Management IP",
        "Serial Number",
        "MAC Address",
        "Ethernet MAC Address",
        "Platform ID",
        "Software Version",
        "Role"
```

```
    ]);

    for device in devices {
        table.add_row(row![
            device.hostname.unwrap_or_else(|| "N/A".to_string()),
            device.management_ip_address.unwrap_or_else(|| "N/A".to_string(
            device.serial_number.unwrap_or_else(|| "N/A".to_string()),
            device.mac_address.unwrap_or_else(|| "N/A".to_string()),
            device.ap_ethernet_mac_address.unwrap_or_else(|| "N/A".to_strin
            device.platform_id.unwrap_or_else(|| "N/A".to_string()),
            device.software_version.unwrap_or_else(|| "N/A".to_string()),
            device.role.unwrap_or_else(|| "N/A".to_string()),
        ]);
    }

    table.printstd();
}

// Function to print detailed information about a device
pub fn print_device_detail(device: AllDevices) {
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Hostname", device.hostname);
    add_field(
        &mut table,
        "Management IP",
        device.management_ip_address,
    );
    add_field(&mut table, "Serial Number", device.serial_number);
    add_field(&mut table, "MAC Address", device.mac_address);
    add_field(&mut table, "Platform ID", device.platform_id);
    add_field(&mut table, "Software Version", device.software_version);
    add_field(&mut table, "Role", device.role);
    add_field(&mut table, "Reachability Status", device.reachability_status
    add_field(&mut table, "Uptime", device.up_time);
    add_field(&mut table, "Last Updated", device.last_update_time.map(|time
        let datetime = DateTime::from_timestamp_millis(timestamp as i64)
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expect("REASO
        datetime.format("%Y-%m-%d %H:%M:%S").to_string()
    }));
    // Add more fields as necessary

    table.printstd();
}

// Function to print enriched device details
pub fn print_device_enrichment(device_details: DeviceDetails) {
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Hostname", device_details.hostname);
```

```
    add_field(
        &mut table,
        "Management IP",
        device_details.managementIpAddress,
    );
    add_field(&mut table, "Serial Number", device_details.serialNumber);
    add_field(&mut table, "MAC Address", device_details.macAddress);
    add_field(
        &mut table,
        "Platform ID",
        device_details.platformId,
    );
    add_field(
        &mut table,
        "Software Version",
        device_details.softwareVersion,
    );
    add_field(
        &mut table,
        "Reachability Status",
        device_details.reachabilityStatus,
    );
    add_field(
        &mut table,
        "Error Code",
        device_details.errorCode.map(|v| v.to_string()),
    );
    add_field(
        &mut table,
        "Error Description",
        device_details.errorDescription,
    );
    // Add more fields as necessary

    table.printstd();
}

// Function to print client detail with all fields
pub fn print_client_detail(response: ClientDetailResponse) {
    if let Some(detail) = response.detail {
        let mut table = Table::new();
        table.add_row(row!["Field", "Value"]);

        add_field(&mut table, "ID", detail.id);
        add_field(&mut table, "Connection Status", detail.connectionStatus);
        add_field(&mut table, "Host Type", detail.hostType);
        add_field(&mut table, "User ID", detail.userId);
        add_field(&mut table, "Host Name", detail.hostName);
        add_field(&mut table, "Host OS", detail.hostOs);
        add_field(&mut table, "Host Version", detail.hostVersion);
        add_field(&mut table, "Sub Type", detail.subType);
```

```
// lastUpdated as timestamp
if let Some(timestamp) = detail.lastUpdated {
    let datetime = DateTime::from_timestamp_millis(timestamp as i64
        .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expect("R
    add_field(
        &mut table,
        "Last Updated",
        Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
    );
} else {
    add_field(&mut table, "Last Updated", None);
}

// Health Score
if let Some(health_scores) = detail.healthScore {
    for (i, hs) in health_scores.iter().enumerate() {
        let prefix = format!("Health Score [{}]", i + 1);
        add_field(&mut table, &format!("{} - Health Type", prefix),
        add_field(&mut table, &format!("{} - Reason", prefix), hs.r
        add_field(
            &mut table,
            &format!("{} - Score", prefix),
            hs.score.map(|s| s.to_string()),
        );
    }
}

add_field(&mut table, "Host MAC", detail.hostMac);
add_field(&mut table, "Host IPv4", detail.hostIpV4);
add_field(
    &mut table,
    "Host IPv6",
    detail.hostIpV6.map(|ips| ips.join(", ")),
);
add_field(&mut table, "Auth Type", detail.authType);
add_field(
    &mut table,
    "VLAN ID",
    detail.vlanId.map(|v| v.to_string()),
);
add_field(
    &mut table,
    "VNID",
    detail.vnid.map(|v| v.to_string()),
);
add_field(&mut table, "SSID", detail.ssid);
add_field(&mut table, "Frequency", detail.frequency);
add_field(&mut table, "Channel", detail.channel);
add_field(&mut table, "AP Group", detail.apGroup);
add_field(&mut table, "Location", detail.location);
add_field(&mut table, "Client Connection", detail.clientConnection)
```

```
// Connected Devices
if let Some(connected_devices) = detail.connectedDevice {
    for (i, cd) in connected_devices.iter().enumerate() {
        let prefix = format!("Connected Device [{}]", i + 1);
        add_field(&mut table, &format!("{} - Type", prefix), cd.dev
        add_field(&mut table, &format!("{} - Name", prefix), cd.nam
        add_field(&mut table, &format!("{} - MAC", prefix), cd.mac.
        add_field(&mut table, &format!("{} - ID", prefix), cd.id.cl
        add_field(&mut table, &format!("{} - IP Address", prefix),
        add_field(&mut table, &format!("{} - Mgmt IP", prefix), cd.
        add_field(&mut table, &format!("{} - Band", prefix), cd.ban
        add_field(&mut table, &format!("{} - Mode", prefix), cd.mod
    }
}

add_field(
    &mut table,
    "Issue Count",
    detail.issueCount.map(|v| v.to_string()),
);
add_field(&mut table, "RSSI", detail.rssi);
add_field(&mut table, "Average RSSI", detail.avgRssi);
add_field(&mut table, "SNR", detail.snr);
add_field(&mut table, "Average SNR", detail.avgSnr);
add_field(&mut table, "Data Rate", detail.dataRate);
add_field(&mut table, "TX Bytes", detail.txBytes);
add_field(&mut table, "RX Bytes", detail.rxBytes);

// Onboarding
if let Some(onboarding) = detail.onboarding {
    // Timestamps
    if let Some(timestamp) = onboarding.authDoneTime {
        let datetime = DateTime::from_timestamp_millis(timestamp as
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expec
        add_field(
            &mut table,
            "Onboarding - Auth Done Time",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Onboarding - Auth Done Time", None);
    }

    if let Some(timestamp) = onboarding.assocDoneTime {
        let datetime = DateTime::from_timestamp_millis(timestamp as
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expec
        add_field(
            &mut table,
            "Onboarding - Assoc Done Time",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
```

```
        add_field(&mut table, "Onboarding - Assoc Done Time", None)
    }

    if let Some(timestamp) = onboarding.dhcpDoneTime {
        let datetime = DateTime::from_timestamp_millis(timestamp as
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expec
        add_field(
            &mut table,
            "Onboarding - DHCP Done Time",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Onboarding - DHCP Done Time", None);
    }

    // Other onboarding fields
    add_field(
        &mut table,
        "Onboarding - Average Run Duration",
        onboarding.averageRunDuration,
    );
    add_field(
        &mut table,
        "Onboarding - Max Run Duration",
        onboarding.maxRunDuration,
    );
    // Add more onboarding fields as necessary

    // Root cause lists
    if let Some(assoc_rc_list) = onboarding.assocRootcauseList {
        add_field(
            &mut table,
            "Onboarding - Assoc Rootcause List",
            Some(assoc_rc_list.join(", ")),
        );
    }
    if let Some(aaa_rc_list) = onboarding.aaaRootcauseList {
        add_field(
            &mut table,
            "Onboarding - AAA Rootcause List",
            Some(aaa_rc_list.join(", ")),
        );
    }
    if let Some(dhcp_rc_list) = onboarding.dhcpRootcauseList {
        add_field(
            &mut table,
            "Onboarding - DHCP Rootcause List",
            Some(dhcp_rc_list.join(", ")),
        );
    }
    if let Some(other_rc_list) = onboarding.otherRootcauseList {
        add_field(
```

```
                &mut table,
                "Onboarding - Other Rootcause List",
                Some(other_rc_list.join(", ")),
            );
        }
        if let Some(latest_rc_list) = onboarding.latestRootCauseList {
            add_field(
                &mut table,
                "Onboarding - Latest Rootcause List",
                Some(latest_rc_list.join(", ")),
            );
        }
    }


    table.printstd();
} else {
    println!("No client details available.");
}

// Optionally, print ConnectionInfo and Topology
if let Some(connection_info) = response.connectionInfo {
    println!("\nConnection Info:");
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Host Type", connection_info.hostType);
    add_field(&mut table, "Network Device Name", connection_info.nwDevi
    add_field(&mut table, "Network Device MAC", connection_info.nwDevic
    add_field(&mut table, "Protocol", connection_info.protocol);
    add_field(&mut table, "Band", connection_info.band);
    add_field(&mut table, "Spatial Stream", connection_info.spatialStre
    add_field(&mut table, "Channel", connection_info.channel);
    add_field(&mut table, "Channel Width", connection_info.channelWidth
    add_field(&mut table, "WMM", connection_info.wmm);
    add_field(&mut table, "UAPSD", connection_info.uapsd);

    // Timestamp
    if let Some(timestamp) = connection_info.timestamp {
        let datetime = DateTime::from_timestamp_millis(timestamp as i64
            .unwrap_or_else(|| DateTime::from_timestamp(0, 0).expect("R
        add_field(
            &mut table,
            "Timestamp",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string()),
        );
    } else {
        add_field(&mut table, "Timestamp", None);
    }

    table.printstd();
}
```

```
if let Some(topology) = response.topology {
    println!("\nTopology Information:");
    // You can choose to display topology data as needed
    if let Some(nodes) = topology.nodes {
        for node in nodes {
            let mut table = Table::new();
            table.add_row(row!["Node Field", "Value"]);
            add_field(&mut table, "Role", node.role);
            add_field(&mut table, "Name", node.name);
            add_field(&mut table, "ID", node.id);
            add_field(&mut table, "Description", node.description);
            add_field(&mut table, "Device Type", node.deviceType);
            add_field(&mut table, "Platform ID", node.platformId);
            add_field(&mut table, "Family", node.family);
            add_field(&mut table, "IP", node.ip);
            add_field(&mut table, "Software Version", node.softwareVers
            add_field(&mut table, "User ID", node.userId);
            add_field(&mut table, "Node Type", node.nodeType);
            add_field(&mut table, "Radio Frequency", node.radioFrequenc
            add_field(
                &mut table,
                "Clients",
                node.clients.map(|v| v.to_string()),
            );
            add_field(
                &mut table,
                "Count",
                node.count.map(|v| v.to_string()),
            );
            add_field(
                &mut table,
                "Health Score",
                node.healthScore.map(|v| v.to_string()),
            );
            add_field(
                &mut table,
                "Level",
                node.level.map(|v| v.to_string()),
            );
            add_field(&mut table, "Fabric Group", node.fabricGroup);
            add_field(&mut table, "Connected Device", node.connectedDev
            if let Some(fabric_roles) = node.fabricRole {
                add_field(
                    &mut table,
                    "Fabric Roles",
                    Some(fabric_roles.join(", ")),
                );
            }
            if let Some(ipv6_list) = node.ipv6 {
                add_field(&mut table, "IPv6", Some(ipv6_list.join(", ")
            }
```

```
                table.printstd();
            }
        }

        // Similarly, you can display links if needed
    }
}

// Helper function to add a field to the table
fn add_field(table: &mut Table, field_name: &str, value: Option<String>) {
    table.add_row(row![
        field_name,
        value.unwrap_or_else(|| "N/A".to_string())
    ]);
}

pub fn print_issue_list(response: IssueListResponse) {
    if let Some(issues) = response.response {
        let mut table = Table::new();
        table.add_row(row![
            "Issue ID",
            "Name",
            "Device ID",
            "Device Role",
            "Client MAC",
            "Status",
            "Priority",
            "Category",
            "Last Occurrence Time"
        ]);

        for issue in issues {
            let last_occurrence = issue.last_occurence_time.map_or("N/A".to
                DateTime::from_timestamp_millis(timestamp)
                    .map(|dt| dt.format("%Y-%m-%d %H:%M:%S").to_string())
                    .unwrap_or_else(|| "Invalid Timestamp".to_string())
            });

            table.add_row(row![
                issue.issueId.clone().unwrap_or_else(|| "N/A".to_string()),
                issue.name.clone().unwrap_or_else(|| "N/A".to_string()),
                issue.deviceId.clone().unwrap_or_else(|| "N/A".to_string())
                issue.deviceRole.clone().unwrap_or_else(|| "N/A".to_string(
                issue.clientMac.clone().unwrap_or_else(|| "N/A".to_string()
                issue.status.clone().unwrap_or_else(|| "N/A".to_string()),
                issue.priority.clone().unwrap_or_else(|| "N/A".to_string())
                issue.category.clone().unwrap_or_else(|| "N/A".to_string())
                last_occurrence,
            ]);
        }
```

```
        table.printstd();
    } else {
        println!("No issues found.");
    }
}

// Function to print AP configuration
pub fn print_ap_config(ap_config: ApConfig) {
    let mut table = Table::new();
    table.add_row(row!["Field", "Value"]);

    add_field(&mut table, "Instance UUID", ap_config.instanceUuid.map(|v| v
    add_field(&mut table, "Instance ID", ap_config.instanceId.map(|v| v.to_
    add_field(&mut table, "Display Name", ap_config.displayName);
    add_field(&mut table, "Instance Tenant ID", ap_config.instanceTenantId)
    add_field(
        &mut table,
        "Ordered List OE Index",
        ap_config._orderedListOEIndex.map(|v| v.to_string()),
    );
    add_field(
        &mut table,
        "Creation Order Index",
        ap_config._creationOrderIndex.map(|v| v.to_string()),
    );
    add_field(
        &mut table,
        "Is Being Changed",
        ap_config._isBeingChanged.map(|v| v.to_string()),
    );
    add_field(&mut table, "Deploy Pending", ap_config.deployPending);
    add_field(&mut table, "Instance Version", ap_config.instanceVersion.map
    add_field(&mut table, "Admin Status", ap_config.adminStatus);
    add_field(&mut table, "AP Height", ap_config.apHeight.map(|v| v.to_stri
    add_field(&mut table, "AP Mode", ap_config.apMode);
    add_field(&mut table, "AP Name", ap_config.apName);
    add_field(&mut table, "Ethernet MAC", ap_config.ethMac);
    add_field(&mut table, "Failover Priority", ap_config.failoverPriority);
    add_field(
        &mut table,
        "LED Brightness Level",
        ap_config.ledBrightnessLevel.map(|v| v.to_string()),
    );
    add_field(&mut table, "LED Status", ap_config.ledStatus);
    add_field(&mut table, "Location", ap_config.location);
    add_field(&mut table, "MAC Address", ap_config.macAddress);
    add_field(&mut table, "Primary Controller Name", ap_config.primaryContr
    add_field(&mut table, "Primary IP Address", ap_config.primaryIpAddress)
    add_field(&mut table, "Secondary Controller Name", ap_config.secondaryC
    add_field(&mut table, "Secondary IP Address", ap_config.secondaryIpAddr
    add_field(&mut table, "Tertiary Controller Name", ap_config.tertiaryCon
    add_field(&mut table, "Tertiary IP Address", ap_config.tertiaryIpAddres
```

```
// Internal Key
if let Some(internal_key) = ap_config.internalKey {
    add_field(&mut table, "Internal Key - Type", internal_key.type_fiel
    add_field(&mut table, "Internal Key - ID", internal_key.id.map(|v|
    add_field(&mut table, "Internal Key - Long Type", internal_key.long
    add_field(&mut table, "Internal Key - URL", internal_key.url);
}

// Display the table
table.printstd();

// Mesh DTOs - Since the schema shows as an array of empty objects, we

// Radio DTOs
if let Some(radio_dtos) = ap_config.radioDTOs {
    for (i, radio) in radio_dtos.iter().enumerate() {
        println!("\nRadio DTO [{}]:", i + 1);
        let mut radio_table = Table::new();
        radio_table.add_row(row!["Field", "Value"]);

        add_field(&mut radio_table, "Display Name", radio.displayName.c
        add_field(&mut radio_table, "Instance ID", radio.instanceId.map
        add_field(
            &mut radio_table,
            "Ordered List OE Index",
            radio._orderedListOEIndex.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Creation Order Index",
            radio._creationOrderIndex.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
            "Is Being Changed",
            radio._isBeingChanged.map(|v| v.to_string()),
        );
        add_field(&mut radio_table, "Deploy Pending", radio.deployPendi
        add_field(
            &mut radio_table,
            "Instance Version",
            radio.instanceVersion.map(|v| v.to_string()),
        );
        add_field(&mut radio_table, "Admin Status", radio.adminStatus.c
        add_field(
            &mut radio_table,
            "Antenna Angle",
            radio.antennaAngle.map(|v| v.to_string()),
        );
        add_field(
            &mut radio_table,
```

```
        "Antenna Elevation Angle",
        radio.antennaElevAngle.map(|v| v.to_string()),
    );
    add_field(
        &mut radio_table,
        "Antenna Gain",
        radio.antennaGain.map(|v| v.to_string()),
    );
    add_field(
        &mut radio_table,
        "Antenna Pattern Name",
        radio.antennaPatternName.clone(),
    );
    add_field(
        &mut radio_table,
        "Channel Assignment Mode",
        radio.channelAssignmentMode.clone(),
    );
    add_field(
        &mut radio_table,
        "Channel Number",
        radio.channelNumber.map(|v| v.to_string()),
    );
    add_field(
        &mut radio_table,
        "Channel Width",
        radio.channelWidth.clone(),
    );
    add_field(&mut radio_table, "Clean Air SI", radio.cleanAirSI.cl
    add_field(&mut radio_table, "Interface Type", radio.ifType.map(
    add_field(
        &mut radio_table,
        "Interface Type Value",
        radio.ifTypeValue.clone(),
    );
    add_field(&mut radio_table, "MAC Address", radio.macAddress.clo
    add_field(
        &mut radio_table,
        "Power Assignment Mode",
        radio.powerAssignmentMode.clone(),
    );
    add_field(
        &mut radio_table,
        "Power Level",
        radio.powerlevel.map(|v| v.to_string()),
    );
    // radioBand and radioRoleAssignment are Option<serde_json::Val
    add_field(
        &mut radio_table,
        "Radio Band",
        radio.radioBand.as_ref().map(|v| v.to_string()),
    );
```

```
            add_field(
                &mut radio_table,
                "Radio Role Assignment",
                radio.radioRoleAssignment.as_ref().map(|v| v.to_string()),
            );
            add_field(&mut radio_table, "Slot ID", radio.slotId.map(|v| v.t

            // Internal Key for RadioDTO
            if let Some(radio_internal_key) = &radio.internalKey {
                add_field(
                    &mut radio_table,
                    "Internal Key - Type",
                    radio_internal_key.type_field.clone(),
                );
                add_field(
                    &mut radio_table,
                    "Internal Key - ID",
                    radio_internal_key.id.map(|v| v.to_string()),
                );
                add_field(
                    &mut radio_table,
                    "Internal Key - Long Type",
                    radio_internal_key.longType.clone(),
                );
                add_field(
                    &mut radio_table,
                    "Internal Key - URL",
                    radio_internal_key.url.clone(),
                );
            }

            // Display the radio table
            radio_table.printstd();
        }
    }
}



pub fn print_client_enrichment(response: ClientEnrichmentResponse) {
    println!("Number of enrichment records: {}", response.0.len());

    for enrichment in response.0 {
        // User Details
        if let Some(user_details) = enrichment.userDetails {
            println!("User Details:");
            let mut table = Table::new();
            table.add_row(row!["Field", "Value"]);

            add_field(&mut table, "ID", user_details.id);
            add_field(&mut table, "Connection Status", user_details.connect
            add_field(&mut table, "Host Type", user_details.hostType);
```

```
add_field(&mut table, "User ID", user_details.userId);
add_field(&mut table, "Host Name", user_details.hostName);
add_field(&mut table, "Host OS", user_details.hostOs);
add_field(&mut table, "Host Version", user_details.hostVersion)
add_field(&mut table, "Sub Type", user_details.subType);

if let Some(timestamp) = user_details.lastUpdated {
    if let Some(datetime) = DateTime::from_timestamp_millis(tim
        add_field(
            &mut table,
            "Last Updated",
            Some(datetime.format("%Y-%m-%d %H:%M:%S").to_string
        );
    } else {
        add_field(&mut table, "Last Updated", Some("Invalid Tim
    }
} else {
    add_field(&mut table, "Last Updated", None);
}

// Health Scores
if let Some(health_scores) = user_details.healthScore {
    for (i, hs) in health_scores.iter().enumerate() {
        let prefix = format!("Health Score [{}]", i + 1);
        add_field(&mut table, &format!("{} - Health Type", pref
        add_field(&mut table, &format!("{} - Reason", prefix),
        add_field(
            &mut table,
            &format!("{} - Score", prefix),
            hs.score.map(|s| s.to_string()),
        );
    }
}

add_field(&mut table, "Host MAC", user_details.hostMac);
add_field(&mut table, "Host IPv4", user_details.hostIpV4);
add_field(
    &mut table,
    "Host IPv6",
    user_details.hostIpV6.map(|ips| ips.join(", ")),
);
add_field(&mut table, "Auth Type", user_details.authType);

// Handling vlanId that can be a string or a number
match user_details.vlanId {
    Some(VlanId::String(ref vlan)) => {
        add_field(&mut table, "VLAN ID", Some(vlan.clone()));
    }
    Some(VlanId::Number(vlan)) => {
        add_field(&mut table, "VLAN ID", Some(vlan.to_string())
    }
    None => {
```

```
            add_field(&mut table, "VLAN ID", None);
        }
    }

    add_field(&mut table, "SSID", user_details.ssid);
    add_field(&mut table, "Location", user_details.location);
    add_field(&mut table, "Client Connection", user_details.clientC

    // Handling issueCount that can be a string or a number
    match user_details.issueCount {
        Some(StringOrNumber::String(ref count)) => {
            add_field(&mut table, "Issue Count", Some(count.clone()
        }
        Some(StringOrNumber::Number(count)) => {
            add_field(&mut table, "Issue Count", Some(count.to_stri
        }
        None => {
            add_field(&mut table, "Issue Count", None);
        }
    }

    // Handling RSSI
    match user_details.rssi {
        Some(StringOrNumber::String(ref value)) => {
            add_field(&mut table, "RSSI", Some(value.clone()));
        }
        Some(StringOrNumber::Number(value)) => {
            add_field(&mut table, "RSSI", Some(value.to_string()));
        }
        None => {
            add_field(&mut table, "RSSI", None);
        }
    }

    // Handling SNR
    match user_details.snr {
        Some(StringOrNumber::String(ref value)) => {
            add_field(&mut table, "SNR", Some(value.clone()));
        }
        Some(StringOrNumber::Number(value)) => {
            add_field(&mut table, "SNR", Some(value.to_string()));
        }
        None => {
            add_field(&mut table, "SNR", None);
        }
    }

    // Handling Data Rate
    match user_details.dataRate {
        Some(StringOrNumber::String(ref value)) => {
            add_field(&mut table, "Data Rate", Some(value.clone()))
        }
```

```
        Some(StringOrNumber::Number(value)) => {
            add_field(&mut table, "Data Rate", Some(value.to_string
        }
        None => {
            add_field(&mut table, "Data Rate", None);
        }
    }

    add_field(&mut table, "Port", user_details.port);

    table.printstd();

    // Onboarding Details
    if let Some(onboarding) = user_details.onboarding {
        println!("Onboarding Details:");
        let mut table = Table::new();
        table.add_row(row!["Field", "Value"]);

        add_field(&mut table, "Average Run Duration", onboarding.av
        add_field(&mut table, "Max Run Duration", onboarding.maxRun
        add_field(&mut table, "Average Assoc Duration", onboarding.
        add_field(&mut table, "Max Assoc Duration", onboarding.maxA
        add_field(&mut table, "Average Auth Duration", onboarding.a
        add_field(&mut table, "Max Auth Duration", onboarding.maxAu
        add_field(&mut table, "Average DHCP Duration", onboarding.a
        add_field(&mut table, "Max DHCP Duration", onboarding.maxDh
        add_field(&mut table, "AAA Server IP", onboarding.aaaServer
        add_field(&mut table, "DHCP Server IP", onboarding.dhcpServ

        // Convert timestamps
        if let Some(auth_done_time) = onboarding.authDoneTime {
            if let Some(datetime) = DateTime::from_timestamp_millis
                add_field(
                    &mut table,
                    "Auth Done Time",
                    Some(datetime.format("%Y-%m-%d %H:%M:%S").to_st
                );
            } else {
                add_field(&mut table, "Auth Done Time", Some("Inval
            }
        }

        if let Some(assoc_done_time) = onboarding.assocDoneTime {
            if let Some(datetime) = DateTime::from_timestamp_millis
                add_field(
                    &mut table,
                    "Assoc Done Time",
                    Some(datetime.format("%Y-%m-%d %H:%M:%S").to_st
                );
            } else {
                add_field(&mut table, "Assoc Done Time", Some("Inva
            }
```

```
        }

        if let Some(dhcp_done_time) = onboarding.dhcpDoneTime {
            if let Some(datetime) = DateTime::from_timestamp_millis
                add_field(
                    &mut table,
                    "DHCP Done Time",
                    Some(datetime.format("%Y-%m-%d %H:%M:%S").to_st
                );
            } else {
                add_field(&mut table, "DHCP Done Time", Some("Inval
            }
        }

        // Handle `latestRootCauseList`
        if let Some(root_causes) = onboarding.latestRootCauseList {
            add_field(&mut table, "Latest Root Causes", Some(root_c
        }

        table.printstd();
    }

    // Connected Devices in UserDetails
    if let Some(connected_devices) = user_details.connectedDevice {
        for (i, conn_dev) in connected_devices.iter().enumerate() {
            println!("\nConnected Device [{}]:", i + 1);
            let mut table = Table::new();
            table.add_row(row!["Field", "Value"]);

            add_field(&mut table, "Type", conn_dev.type_field.clone
            add_field(&mut table, "Name", conn_dev.name.clone());
            add_field(&mut table, "MAC", conn_dev.mac.clone());
            add_field(&mut table, "ID", conn_dev.id.clone());
            add_field(&mut table, "IP Address", conn_dev.ip_address
            add_field(&mut table, "Mgmt IP", conn_dev.mgmtIp.clone(
            add_field(&mut table, "Band", conn_dev.band.clone());
            add_field(&mut table, "Mode", conn_dev.mode.clone());

            table.printstd();
        }
    }
} else {
    println!("User Details Missing");
}

// Connected Devices in Enrichment
if let Some(connected_devices) = enrichment.connectedDevice {
    for (i, conn_dev) in connected_devices.iter().enumerate() {
        if let Some(device_details) = &conn_dev.deviceDetails {
            println!("\nConnected Device [{}]:", i + 1);
            let mut table = Table::new();
            table.add_row(row!["Field", "Value"]);
```

```
            add_field(&mut table, "Family", device_details.family.c
            add_field(&mut table, "Type", device_details.type_field
            add_field(&mut table, "Location", device_details.locati
            add_field(&mut table, "Error Code", device_details.erro
            add_field(&mut table, "MAC Address", device_details.mac
            add_field(&mut table, "Role", device_details.role.clone
            add_field(
                &mut table,
                "AP Manager Interface IP",
                device_details.apManagerInterfaceIp.clone(),
            );
            add_field(
                &mut table,
                "Associated WLC IP",
                device_details.associatedWlcIp.clone(),
            );
            add_field(&mut table, "Boot Date Time", device_details.
            add_field(
                &mut table,
                "Collection Status",
                device_details.collectionStatus.clone(),
            );
            // Add more fields as needed

            table.printstd();
        }
    }
} else {
    println!("Connected Devices Missing");
}

// Issue Details
if let Some(issue_details) = enrichment.issueDetails {
    if let Some(issues) = issue_details.issue {
        for (i, issue) in issues.iter().enumerate() {
            println!("\nIssue [{}]:", i + 1);
            let mut table = Table::new();
            table.add_row(row!["Field", "Value"]);

            add_field(&mut table, "Issue ID", issue.issueId.clone()
            add_field(&mut table, "Issue Source", issue.issueSource
            add_field(&mut table, "Issue Category", issue.issueCate
            add_field(&mut table, "Issue Name", issue.issueName.clo
            add_field(&mut table, "Issue Description", issue.issueD
            add_field(&mut table, "Issue Entity", issue.issueEntity
            add_field(&mut table, "Issue Entity Value", issue.issue
            add_field(&mut table, "Issue Severity", issue.issueSeve
            add_field(&mut table, "Issue Priority", issue.issuePrio
            add_field(&mut table, "Issue Summary", issue.issueSumma
            if let Some(timestamp) = issue.issueTimestamp {
                if let Some(datetime) = DateTime::from_timestamp_mi
```

```
                            add_field(
                                &mut table,
                                "Issue Timestamp",
                                Some(datetime.format("%Y-%m-%d %H:%M:%S").t
                            );
                        } else {
                            add_field(&mut table, "Issue Timestamp", Some("
                        }
                    } else {
                        add_field(&mut table, "Issue Timestamp", None);
                    }
                    // Handle suggestedActions and impactedHosts if necessa

                    table.printstd();
                }
            } else {
                println!("No Issues Found");
            }
        } else {
            println!("Issue Details Missing");
        }
    }
}
```