RVI20-64 Processor Certification Requirements Document

Table of Contents

CRD Revision History . . . 1
Typographic Conventions . . . 1
Glossary . . . 2
1. Introduction . . . 2
2. Extensions . . . 5
3. Implementation-dependencies . . . 6
4. Traps . . . 6
5. Instruction Summary . . . 7
6. CSR Summary . . . 11
Appendix A: Extension Details . . . 14
Appendix B: Instruction Details . . . 32
Appendix C: CSR Details . . . 449
Appendix D: IDL Function Details . . . 537

CRD Revision History

History of documentation changes that eventually lead to releases.

Date	Revision	Changes
2025-07-03	0.1.0	• Created

Typographic Conventions

CSR field colors

- Grey fields are reserved (WPRI)
- Green fields are present
- Red fields are defined by the RISC-V ISA but not present

CSR field types

Abbreviation	Description
RO	**Read-Only** Field has a hardwired value that does not change. Writes to an RO field are ignored.
RO-H	**Read-Only with Hardware update** Writes are ignored. Reads reflect a value dynamically generated by hardware.
RW	**Read-Write** Field is writable by software. Any value that fits in the field is acceptable and shall be retained for subsequent reads.
RW-R	**Read-Write Restricted** Field is writable by software. Only certain values are legal. Writing an illegal value into the field is ignored, and the field retains its prior state.
RW-H	**Read-Write with Hardware update** Field is writable by software. Any value that fits in the field is acceptable. Hardware also updates the field without an explicit software write.
RW-RH	**Read-Write Restricted with Hardware update** Field is writable by software. Only certain values are legal. Writing an illegal value into the field is ignored, such that the field retains its prior state. Hardware also updates the field without an explicit software write.)

Glossary

Table 1. Glossary

Term	Meaning
RISC-V	(Reduced Instruction Set Computer) architecture, version 5
RVI	RISC-V International (organization that oversees RISC-V)
RVCP	RISC-V Certification Program
TSC	Technical Steering Committee (branch of RVI that creates standards)
CSC	Certification Steering Committee (branch of RVI that oversees the RVCP)
CRD	Certification Requirements Document
CTP	Certification Test Plan
CSR	Control & Status Register (located inside processor, not memory-mapped)
TBD	To Be Determined
N/A	“Not Applicable”
AKA	“Also Known As”
Must	Indicates a mandatory requirement
Should	Indicates a recommended requirement
May	Indicates an optional requirement

1. Introduction

The RVI20-64 processor certificate corresponds to the RVI20U64 profile.

The RVI certificate class corresponds to the RVI Profile Family. This certificate class only includes the RISC-V Unprivileged ISA. Certificates for the RVI certificate class are intended for internal use by the RVCP (RISC-V Certification Program).

1.1. What’s a CRD?

Certification Requirements Documents (CRDs) list requirements an implementation must meet to obtain an associated RVI (RISC-V International) certificate. CRDs are developed by the RVI CSC (Certification Steering Committee) organization in collaboration with the RVI TSC (Technical Steering Committee) organization who creates RISC-V standards.

The CRDs refer to and augment information provided in existing ratified RVI standards.

There are a variety of certificates offered by RVI to accomodate the various RVI standards. There are certificates for processors, non-processor system IP (e.g., IOMMU), and system platforms (processor + system IP) hardware standards. There are multiple classes of processor certificates available to accomodate the wide range of RISC-V implementations from basic microcontrollers to advanced Applications-class processors.

Each CRD has a list of mandatory behaviors along with a list of optional behaviors. Note that not all behaviors allowed in RISC-V standards are supported by a particular CRD.

1.2. Naming Scheme

1.2.1. CRD Naming

All components of the RVCP share the following naming scheme:

```
Format: <name>[v<version>]
```

Where:

- Left & right square braces denote optional.
- Less-than & greater-than signs just separate fields (i.e., they aren’t present in the CRD name).
- <name> identifies the type of RISC-V standard (processor, non-processor system IP, or platform) along with any other information required to identify the variant of that standard.
- <version> identifies a particular release
  - Format is <major>[.<minor>[.<patch>]]
  - Inspired by semantic versioning scheme (semver.org/) but doesn’t follow it exactly

The rules for updating <major>, <minor>, and <patch> are defined by the type of RVCP component. However, the follow these general guidelines:

- A <major> release of 0 is used for pre-release versions and release versions start with 1.
- The <major> release number is updated when mandatory changes are made.
- The <minor> release number is updated when optional changes are made.
- The <patch> release number is updated for documentation fixes/improvements that don't affect certificates already obtained for a particular implementation.

The specific rules for updating the version number of a CRD are as follows:

- The <major> release number is updated for changes that **could** cause a previously certified implementation to no longer meet requirements. An example is requiring a new version of a standard.
- The <minor> release number is updated for increased support of optional behaviors.
- The <patch> release number is updated for documentation fixes/improvements. These changes **cannot** cause a previously certified implementation to no longer meet requirements.

1.2.2. Processor Naming

RVCP components for processors have the following format for their <name>:

```
<class><model>[<-base>]
```

Where:

- <class> is MC for Microcontroller Class and AC for Apps-processor Class
- <model> is 3-digit integer defined as follows:
  - The hundreds's digit indicates the series
  - The ten's digit identifies large differences in mandatory extensions (e.g., V, H) within the series
  - The one's digit indentifies small/medium differences in mandatory extensions (e.g., Zicond, PMP) within the series
- <base> is optional and is 32 for RV32I, 64 for RV64I, and 32E for RV32E
  - If multiple bases are supported and <base> is omitted in a reference, the reference applies to all bases
  - If only one base is supported then <base> is generally omitted

1.3. Requirements Terminology

Table 2. Requirement Types

Term	Meaning
MANDATORY	You have to implement it to get a certificate and the certificate tests will cover it
OPTIONAL	It's up to you if you implement or not. If you claim to implement it, certificate tests will cover it
IN-SCOPE	Either MANDATORY or OPTIONAL
OUT-OF-SCOPE	It's up to you if you implement or not. If you implement it, it won't be certified but make sure you don't mess up anything we are certifying.
INCOMPATIBLE	If you implement it you won't get a certificate

Table 3. Definition of CSR Fields

Field Type	Read Value After Writing Illegal Value	Read Value Function Of	Illegal Instruction Exception	Priv ISA Manual Quote
WLRL	Any deterministic legal or illegal value	Value before write and illegal value written	Optional	Implementations are permitted but not required to raise an illegal-instruction exception if an instruction attempts to write a non-supported value to a WLRL field. Implementations can return arbitrary bit patterns on the read of a WLRL field when the last write was of an illegal value, but the value returned should deterministically depend on the illegal written value and the value of the field prior to the write.
WARL	Any deterministic legal value	Any architectural hart state	Prohibited	Implementations will not raise an exception on writes of unsupported values to a WARL field. Implementations can return any legal value on the read of a WARL field when the last write was of an illegal value, but the legal value returned should deterministically depend on the illegal written value and the architectural state of the hart.
WPRI	0	Nothing	Not specified	Some whole read/write fields are reserved for future use. Implementations that do not furnish these fields must make them read-only zero.

WARL (Write Anything, Read Legal):

The Priv ISA requires reads of WARL fields to return some implementation-dependent deterministic legal value after the field is written with an illegal value. Certifying such behaviors is expensive and provides low value for a certificate since software can't rely on a particular behavior from one implementation to another.

Processor CRDs define writes to WARL fields of illegal values to be OUT-OF-SCOPE unless otherwise stated (i.e., certification tests will only ever write legal values to WARL fields except for the special cases listed below). When not OUT-OF-SCOPE, the required behavior is defined as this might be more constrained in implementations than in the standard.

The following special cases for WARL are supported when explicitly listed in the corresponding CRD CSR field requirements:

1. Probing for Field Width
    - Some WARL fields are variable length such as the ASID field in the virtual memory extension.
    - Here's the algorithm recommended to discover the ASID width:
        - The number of implemented ASID bits, termed ASIDLEN, may be determined by writing one to every bit position in the ASID field, then reading back the value in the satp CSR to see which bit positions in the ASID field hold a one.
    - The RVCP-provided certification materials (certification tests, certification reference models) can map writes of illegal values to the ASID field to the corresponding read value as long as they are provided the ASIDLEN value for an implementation.
2. Probing for Options
    - E.g., Writable misa bits
3. Allowed values are a function of extension presence and/or their parameters
    - E.g., satp.mode legal write values

WLRL (Write Legal, Read Legal):

The Priv ISA requires reads of WLRL fields to return some implementation-dependent deterministic arbitrary value after the field is written with an illegal value. Certifying such behaviors is expensive and provides low value for a certificate since software can't rely on a particular behavior. Processor CRDs define writes to WLRL fields of illegal values to be OUT-OF-SCOPE unless otherwise stated (i.e., certification tests will only ever write legal values to WLRL fields).

WPRI (Write Preserve, Read Ignore):

The Priv ISA requires reads of WPRI fields to return a value of 0. Such WPRI fields are always unimplemented by definition. Certification tests are aware of which fields in the CSRs are WPRI and normally write them with 0 but will also write them with ~0 (all ones) and ensure that reads return 0 in both cases. It is OUT-OF-SCOPE for certification tests to write all possible values of WPRI fields (especially if they are more than just a few bits) and certification tests aren't intended to be comprehensive verification test suites anyways.

1.4. Related Specifications

Certificate Model	TSC Profile	Unpriv ISA Manual	Priv ISA Manual	Debug Manual
RVI20-64	RVI20	20240411		

1.5. Privileged Modes

M	S	U	HS	VS	VU
OUT-OF-SCOPE	OUT-OF-SCOPE	IN-SCOPE	OUT-OF-SCOPE	OUT-OF-SCOPE	OUT-OF-SCOPE

2. Extensions

Any RISC-V extensions not listed in this section are OUT-OF-SCOPE. The RVI20-64 certificate doesn't cover their behaviors.

2.1. Mandatory Extensions

The RVI20-64 certificate has 1 mandatory extensions.

Requirement ID	Extension	Version	Long Name	Note
REQ-EXT-I	I	~> 2.1	Base integer ISA (RV32I or RV64I)	RVI is the mandatory base ISA for RVA, and is little-endian. As per the unprivileged architecture specification, the ecall instruction causes a requested trap to the execution environment. Misaligned loads and stores might not be supported. The fence.tso instruction is mandatory. NOTE: The fence.tso instruction was incorrectly described as optional in the 2019 ratified specifications. However, fence.tso is encoded within the standard fence encoding such that implementations must treat it as a simple global fence if they do not natively support TSO-ordering optimizations. As software can always assume without any penalty that fence.tso is being exploited by a hardware implementation, there is no advantage to making the instruction a profile option. Later versions of the unprivileged ISA specifications correctly indicate that fence.tso is mandatory.

2.2. Optional Extensions

The RVI20-64 certificate has 8 optional extensions.

Requirement ID	Extension	Version	Long Name	Note
REQ-EXT-A	A	= 2.1	Atomic instructions	
REQ-EXT-C	C	= 2.0	Compressed instructions	
REQ-EXT-D	D	= 2.2	Double-precision floating-point	NOTE: The rationale to not include Q as a profile option is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect software to expend effort optimizing use of Q instructions in case they are present.
REQ-EXT-F	F	= 2.2	Single-precision floating-point	
REQ-EXT-M	M	= 2.0	Integer multiply and divide instructions	
REQ-EXT-Zicntr	Zicntr	= 2.0	Architectural performance counters	
REQ-EXT-Zifencei	Zifencei	= 2.0	Instruction fence	
REQ-EXT-Zihpm	Zihpm	= 2.0	Programmable hardware performance counters	The number of counters is platform-specific.

3. Implementation-dependencies

RISC-V standards support many implementation-defined parameters. In many cases, there are no names associated with these parameters. Names are defined in this section when not provided in the associated standard.

3.1. IN-SCOPE Parameters

These implementation-dependent options defined by MANDATORY or OPTIONAL extensions are IN-SCOPE. An implementation must abide by the "Allowed Value(s)" to obtain a certificate. If the "Allowed Value(s)" is "Any" then any value allowed by the type is acceptable.

None

3.2. OUT-OF-SCOPE Parameters

These implementation-dependent options defined by MANDATORY or OPTIONAL extensions are OUT-OF-SCOPE. There are no restrictions on their values for certification purposes because the certificate doesn’t cover the behavior of the associated RISC-V standard as a function of these parameters.

The RVI20-64 certificate has 13 OUT-OF-SCOPE parameters.

Parameters	Type	Extension (s)
HW_MSTATUS_FS_DIRTY_UPDATE	[never, precise, imprecise]	F
LRSC_FAIL_ON_NON_EXACT_LRSC	boolean	A
LRSC_FAIL_ON_VA_SYNONYM	boolean	A
LRSC_MISALIGNED_BEHAVIOR	[always raise misaligned exception, always raise access fault, custom]	A
LRSC_RESERVATION_STRATEGY	[reserve naturally-aligned 64-byte region, reserve naturally-aligned 128-byte region, reserve exactly enough to cover the access, custom]	A
MISALIGNED_AMO	boolean	A
MSTATUS_FS_LEGAL_VALUES	at most 4-element array of [0, 1, 2, 3]	F
MUTABLE_MISA_A	boolean	A
MUTABLE_MISA_C	boolean	C
MUTABLE_MISA_D	boolean	D
MUTABLE_MISA_F	boolean	F
MUTABLE_MISA_M	boolean	M
TIME_CSR_IMPLEMENTED	boolean	Zicntr

4. Traps

RISC-V supports both synchronous exceptions and asynchronous interrupts. TODO: List only traps that exist in this processor certificate model (currently lists all possible in present extensions). See github.com/riscv-software-src/riscv-unified-db/issues/291 and github.com/riscv-software-src/riscv-unified-db/issues/324 TODO: Show traps per privilege mode

4.1. Synchronous Exceptions

`xcause.CODE` CSR Field Value	Name

4.2. Asynchronous Interrupts

`xcause.CODE` CSR Field Value	Name
1	Supervisor software interrupt
2	Virtual supervisor software interrupt
3	Machine software interrupt
5	Supervisor timer interrupt
6	Virtual supervisor timer interrupt

7	Machine timer interrupt
9	Supervisor external interrupt
10	Virtual supervisor external interrupt
11	Machine external interrupt
12	Supervisor guest external interrupt
13	Local counter overflow interrupt

5. Instruction Summary

The RVI20-64 certificate has up to 225 instructions (exact number depends on an implementation's options).

Name	Long Name
add	Integer add
addi	Add immediate
addiw	Add immediate word
addw	Add word
amoadd.d	Atomic fetch-and-add doubleword
amoadd.d.aq	Atomic fetch-and-add doubleword (acquire)
amoadd.d.aqrl	Atomic fetch-and-add doubleword (acquire-release)
amoadd.d.rl	Atomic fetch-and-add doubleword (release)
amoadd.w	Atomic fetch-and-add word
amoadd.w.aq	Atomic fetch-and-add word (acquire)
amoadd.w.aqrl	Atomic fetch-and-add word (acquire-release)
amoadd.w.rl	Atomic fetch-and-add word (release)
amoand.d	Atomic fetch-and-and doubleword
amoand.d.aq	Atomic fetch-and-and doubleword (acquire)
amoand.d.aqrl	Atomic fetch-and-and doubleword (acquire-release)
amoand.d.rl	Atomic fetch-and-and doubleword (release)
amoand.w	Atomic fetch-and-and word
amoand.w.aq	Atomic fetch-and-and word (acquire)
amoand.w.aqrl	Atomic fetch-and-and word (acquire-release)
amoand.w.rl	Atomic fetch-and-and word (release)
amomax.d	Atomic MAX doubleword
amomax.d.aq	Atomic MAX doubleword (acquire)
amomax.d.aqrl	Atomic MAX doubleword (acquire-release)
amomax.d.rl	Atomic MAX doubleword (release)
amomax.w	Atomic MAX word
amomax.w.aq	Atomic MAX word (acquire)
amomax.w.aqrl	Atomic MAX word (acquire-release)
amomax.w.rl	Atomic MAX word (release)
amomaxu.d	Atomic MAX unsigned doubleword
amomaxu.d.aq	Atomic MAX unsigned doubleword (acquire)
amomaxu.d.aqrl	Atomic MAX unsigned doubleword (acquire-release)
amomaxu.d.rl	Atomic MAX unsigned doubleword (release)
amomaxu.w	Atomic MAX unsigned word
amomaxu.w.aq	Atomic MAX unsigned word (acquire)
amomaxu.w.aqrl	Atomic MAX unsigned word (acquire-release)
amomaxu.w.rl	Atomic MAX unsigned word (release)
amomin.d	Atomic MIN doubleword
amomin.d.aq	Atomic MIN doubleword (acquire)
amomin.d.aqrl	Atomic MIN doubleword (acquire-release)

Name	Long Name
amomin.d.rl	Atomic MIN doubleword (release)
amomin.w	Atomic MIN word
amomin.w.aq	Atomic MIN word (acquire)
amomin.w.aqrl	Atomic MIN word (acquire-release)
amomin.w.rl	Atomic MIN word (release)
amominu.d	Atomic MIN unsigned doubleword
amominu.d.aq	Atomic MIN unsigned doubleword (acquire)
amominu.d.aqrl	Atomic MIN unsigned doubleword (acquire-release)
amominu.d.rl	Atomic MIN unsigned doubleword (release)
amominu.w	Atomic MIN unsigned word
amominu.w.aq	Atomic MIN unsigned word (acquire)
amominu.w.aqrl	Atomic MIN unsigned word (acquire-release)
amominu.w.rl	Atomic MIN unsigned word (release)
amoor.d	Atomic fetch-and-or doubleword
amoor.d.aq	Atomic fetch-and-or doubleword (acquire)
amoor.d.aqrl	Atomic fetch-and-or doubleword (acquire-release)
amoor.d.rl	Atomic fetch-and-or doubleword (release)
amoor.w	Atomic fetch-and-or word
amoor.w.aq	Atomic fetch-and-or word (acquire)
amoor.w.aqrl	Atomic fetch-and-or word (acquire-release)
amoor.w.rl	Atomic fetch-and-or word (release)
amoswap.d	Atomic SWAP doubleword
amoswap.d.aq	Atomic SWAP doubleword (acquire)
amoswap.d.aqrl	Atomic SWAP doubleword (acquire-release)
amoswap.d.rl	Atomic SWAP doubleword (release)
amoswap.w	Atomic SWAP word
amoswap.w.aq	Atomic SWAP word (acquire)
amoswap.w.aqrl	Atomic SWAP word (acquire-release)
amoswap.w.rl	Atomic SWAP word (release)
amoxor.d	Atomic fetch-and-xor doubleword
amoxor.d.aq	Atomic fetch-and-xor doubleword (acquire)
amoxor.d.aqrl	Atomic fetch-and-xor doubleword (acquire-release)
amoxor.d.rl	Atomic fetch-and-xor doubleword (release)
amoxor.w	Atomic fetch-and-xor word
amoxor.w.aq	Atomic fetch-and-xor word (acquire)
amoxor.w.aqrl	Atomic fetch-and-xor word (acquire-release)
amoxor.w.rl	Atomic fetch-and-xor word (release)
and	And
andi	And immediate
auipc	Add upper immediate to pc
beq	Branch if equal
bge	Branch if greater than or equal
bgeu	Branch if greater than or equal unsigned
blt	Branch if less than
bltu	Branch if less than unsigned
bne	Branch if not equal
c.fld	Load double-precision
c.fldsp	Load doubleword into floating-point register from stack
c.flw	Load single-precision

Name	Long Name
c.flwsp	Load word into floating-point register from stack
c.fsd	Store double-precision
c.fsdsp	Store double-precision value to stack
c.fsw	Store single-precision
c.fswsp	Store single-precision value to stack
div	Signed division
divu	Unsigned division
divuw	Unsigned 32-bit division
divw	Signed 32-bit division
ebreak	Breakpoint exception
ecall	Environment call
fadd.d	Floating-Point Add Double-Precision
fadd.s	Floating-Point Add Single-Precision
fclass.d	Floating-Point Classify Double-Precision
fclass.s	Floating-Point Classify Single-Precision
fcvt.d.h	No synopsis available
fcvt.d.l	Floating-Point Convert Long to Double-Precision
fcvt.d.lu	Floating-Point Convert Unsigned Long to Double-Precision
fcvt.d.s	Floating-Point Convert Single-Precision to Double-Precision
fcvt.d.w	Floating-Point Convert Word to Double-Precision
fcvt.d.wu	Floating-Point Convert Unsigned Word to Double-Precision
fcvt.h.d	Floating-point Convert Double-precision to Half-precision
fcvt.l.d	Floating-Point Convert Double-Precision to Long
fcvt.l.s	Floating-Point Convert Single-Precision to Long
fcvt.lu.d	Floating-Point Convert Double-Precision to Unsigned Long
fcvt.lu.s	Floating-Point Convert Single-Precision to Unsigned Long
fcvt.s.d	Floating-Point Convert Double-Precision to Single-Precision
fcvt.s.l	Floating-Point Convert Long to Single-Precision
fcvt.s.lu	Floating-Point Convert Unsigned Long to Single-Precision
fcvt.s.w	Floating-Point Convert Word to Single-Precision
fcvt.s.wu	Floating-Point Convert Unsigned Word to Single-Precision
fcvt.w.d	Floating-Point Convert Double-Precision to Word
fcvt.w.s	Floating-Point Convert Single-Precision to Word
fcvt.wu.d	Floating-Point Convert Double-Precision to Unsigned Word
fcvt.wu.s	Floating-Point Convert Single-Precision to Unsigned Word
fcvtmod.w.d	Floating-Point Convert Double-Precision to Word with Modulo
fdiv.d	Floating-Point Divide Double-Precision
fdiv.s	Floating-Point Divide Single-Precision
fence	Memory ordering fence
fence.i	Instruction fence
fence.tso	Memory ordering fence, total store ordering
feq.d	Floating-Point Equal Double-Precision
feq.s	Floating-Point Equal Single-Precision
fld	Floating-Point Load Double-Precision
fle.d	Floating-Point Less Than or Equal Double-Precision
fle.s	Floating-Point Less Than or Equal Single-Precision
fleq.d	Floating-Point Less Than or Equal Quiet Double-Precision
fli.d	Floating-Point Load Immediate Double-Precision
flt.d	Floating-Point Less Than Double-Precision

Name	Long Name
flt.s	Floating-Point Less Than Single-Precision
fltq.d	Floating-Point Less Than Quiet Double-Precision
flw	Floating-Point Load Single-Precision
fmadd.d	Floating-Point Multiply-Add Double-Precision
fmadd.s	Floating-Point Multiply-Add Single-Precision
fmax.d	Floating-Point Maximum-Number Double-Precision
fmax.s	Floating-Point Maximum-Number Single-Precision
fmaxm.d	Floating-Point Maximum-Number NaN Double-Precision
fmin.d	Floating-Point Minimum-Number Double-Precision
fmin.s	Floating-Point Minimum-Number Single-Precision
fminm.d	Floating-Point Minimum-Number NaN Double-Precision
fmsub.d	Floating-Point Multiply-Subtract Double-Precision
fmsub.s	Floating-Point Multiply-Subtract Single-Precision
fmul.d	Floating-Point Multiply Double-Precision
fmul.s	Floating-Point Multiply Single-Precision
fmv.d.x	Floating-Point Move from Integer Register to Double-Precision Register
fmv.w.x	Floating-Point Move Single-Precision Word from Integer Register
fmv.x.d	Floating-Point Move from Double-Precision Register to Integer Register
fmv.x.w	Floating-Point Move Single-Precision Word to Integer Register
fmvh.x.d	Floating-Point Move High Half from Double-Precision Register to Integer Register
fmvp.d.x	Floating-Point Move Pair from Integer Registers to Double-Precision Register
fnmadd.d	Floating-Point Negate-Multiply-Add Double-Precision
fnmadd.s	Floating-Point Negate-Multiply-Add Single-Precision
fnmsub.d	Floating-Point Negate-Multiply-Subtract Double-Precision
fnmsub.s	Floating-Point Negate-Multiply-Subtract Single-Precision
fround.d	Floating-Point Round Double-Precision
froundnx.d	Floating-Point Round-to-Integer Inexact Double-Precision
fsd	Floating-Point Store Double-Precision
fsgnj.d	Floating-Point Sign-Inject Double-Precision
fsgnj.s	Floating-Point Sign-Inject Single-Precision
fsgnjn.d	Floating-Point Sign-Inject Negate Double-Precision
fsgnjn.s	Floating-Point Sign-Inject Negate Single-Precision
fsgnjx.d	Floating-Point Sign-Inject XOR Double-Precision
fsgnjx.s	Floating-Point Sign-Inject XOR Single-Precision
fsqrt.d	Floating-Point Square Root Double-Precision
fsqrt.s	Floating-Point Square Root Single-Precision
fsub.d	Floating-Point Subtract Double-Precision
fsub.s	Floating-Point Subtract Single-Precision
fsw	Floating-Point Store Single-Precision
jal	Jump and link
jalr	Jump and link register
lb	Load byte
lbu	Load byte unsigned
ld	Load doubleword
lh	Load halfword
lhu	Load halfword unsigned
lr.d	Load reserved doubleword
lr.w	Load reserved word
lui	Load upper immediate

Name	Long Name
lw	Load word
lwu	Load word unsigned
mul	Signed multiply
mulh	Signed multiply high
mulhsu	Signed/unsigned multiply high
mulhu	Unsigned multiply high
mulw	Signed 32-bit multiply
or	Or
ori	Or immediate
rem	Signed remainder
remu	Unsigned remainder
remuw	Unsigned 32-bit remainder
remw	Signed 32-bit remainder
sb	Store byte
sc.d	Store conditional doubleword
sc.w	Store conditional word
sd	Store doubleword
sh	Store halfword
sll	Shift left logical
slli	Shift left logical immediate
slliw	Shift left logical immediate word
sllw	Shift left logical word
slt	Set on less than
slti	Set on less than immediate
sltiu	Set on less than immediate unsigned
sltu	Set on less than unsigned
sra	Shift right arithmetic
srai	Shift right arithmetic immediate
sraiw	Shift right arithmetic immediate word
sraw	Shift right arithmetic word
srl	Shift right logical
srli	Shift right logical immediate
srliw	Shift right logical immediate word
srlw	Shift right logical word
sub	Subtract
subw	Subtract word
sw	Store word
xor	Exclusive Or
xori	Exclusive Or immediate

6. CSR Summary

The RVI20-64 certificate has up to 42 CSRs (exact number depends on an implementation's options).

6.1. By Name

Name	Long Name	Address	Mode	Primary Extension
cycle	Cycle counter for RDCYCLE Instruction	0xc00	U	Zicntr ~> 2.0.0
cycleh	High-half cycle counter for RDCYCLE Instruction	0xc80	U	Zicntr ~> 2.0.0
fcsr	Floating-point control and status register (frm + fflags)	0x3	U	F ~> 2.2.0
fflags	Floating-Point Accrued Exceptions	0x1	U	F ~> 2.2.0

Name	Long Name	Address	Mode	Primary Extension
frm	Floating-Point Dynamic Rounding Mode	0x2	U	F ~> 2.2.0
hpmcounter10	User-mode Hardware Performance Counter 7	0xc0a	U	Zihpm ~> 2.0.0
hpmcounter11	User-mode Hardware Performance Counter 8	0xc0b	U	Zihpm ~> 2.0.0
hpmcounter12	User-mode Hardware Performance Counter 9	0xc0c	U	Zihpm ~> 2.0.0
hpmcounter13	User-mode Hardware Performance Counter 10	0xc0d	U	Zihpm ~> 2.0.0
hpmcounter14	User-mode Hardware Performance Counter 11	0xc0e	U	Zihpm ~> 2.0.0
hpmcounter15	User-mode Hardware Performance Counter 12	0xc0f	U	Zihpm ~> 2.0.0
hpmcounter16	User-mode Hardware Performance Counter 13	0xc10	U	Zihpm ~> 2.0.0
hpmcounter17	User-mode Hardware Performance Counter 14	0xc11	U	Zihpm ~> 2.0.0
hpmcounter18	User-mode Hardware Performance Counter 15	0xc12	U	Zihpm ~> 2.0.0
hpmcounter19	User-mode Hardware Performance Counter 16	0xc13	U	Zihpm ~> 2.0.0
hpmcounter20	User-mode Hardware Performance Counter 17	0xc14	U	Zihpm ~> 2.0.0
hpmcounter21	User-mode Hardware Performance Counter 18	0xc15	U	Zihpm ~> 2.0.0
hpmcounter22	User-mode Hardware Performance Counter 19	0xc16	U	Zihpm ~> 2.0.0
hpmcounter23	User-mode Hardware Performance Counter 20	0xc17	U	Zihpm ~> 2.0.0
hpmcounter24	User-mode Hardware Performance Counter 21	0xc18	U	Zihpm ~> 2.0.0
hpmcounter25	User-mode Hardware Performance Counter 22	0xc19	U	Zihpm ~> 2.0.0
hpmcounter26	User-mode Hardware Performance Counter 23	0xc1a	U	Zihpm ~> 2.0.0
hpmcounter27	User-mode Hardware Performance Counter 24	0xc1b	U	Zihpm ~> 2.0.0
hpmcounter28	User-mode Hardware Performance Counter 25	0xc1c	U	Zihpm ~> 2.0.0
hpmcounter29	User-mode Hardware Performance Counter 26	0xc1d	U	Zihpm ~> 2.0.0
hpmcounter3	User-mode Hardware Performance Counter 0	0xc03	U	Zihpm ~> 2.0.0
hpmcounter30	User-mode Hardware Performance Counter 27	0xc1e	U	Zihpm ~> 2.0.0
hpmcounter31	User-mode Hardware Performance Counter 28	0xc1f	U	Zihpm ~> 2.0.0
hpmcounter4	User-mode Hardware Performance Counter 1	0xc04	U	Zihpm ~> 2.0.0
hpmcounter5	User-mode Hardware Performance Counter 2	0xc05	U	Zihpm ~> 2.0.0
hpmcounter6	User-mode Hardware Performance Counter 3	0xc06	U	Zihpm ~> 2.0.0
hpmcounter7	User-mode Hardware Performance Counter 4	0xc07	U	Zihpm ~> 2.0.0
hpmcounter8	User-mode Hardware Performance Counter 5	0xc08	U	Zihpm ~> 2.0.0
hpmcounter9	User-mode Hardware Performance Counter 6	0xc09	U	Zihpm ~> 2.0.0
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U	Zicntr ~> 2.0.0
instreth	Instructions retired counter, high bits	0xc82	U	Zicntr ~> 2.0.0
mcycle	Machine Cycle Counter	0xb00	M	Zicntr ~> 2.0.0
mcycleh	High-half machine Cycle Counter	0xb80	M	Zicntr ~> 2.0.0
minstret	Machine Instructions Retired Counter	0xb02	M	Zicntr ~> 2.0.0
minstreth	Machine Instructions Retired Counter	0xb82	M	Zicntr ~> 2.0.0
time	Timer for RDTIME Instruction	0xc01	U	Zicntr ~> 2.0.0
timeh	High-half timer for RDTIME Instruction	0xc81	U	Zicntr ~> 2.0.0

6.2. By Address

Address	Mode	Name	Long Name	Primary Extension
0x1	U	fflags	Floating-Point Accrued Exceptions	F ~> 2.2.0
0x2	U	frm	Floating-Point Dynamic Rounding Mode	F ~> 2.2.0
0x3	U	fcsr	Floating-point control and status register (frm + fflags)	F ~> 2.2.0
0xb00	M	mcycle	Machine Cycle Counter	Zicntr ~> 2.0.0
0xb02	M	minstret	Machine Instructions Retired Counter	Zicntr ~> 2.0.0
0xb80	M	mcycleh	High-half machine Cycle Counter	Zicntr ~> 2.0.0
0xb82	M	minstreth	Machine Instructions Retired Counter	Zicntr ~> 2.0.0
0xc00	U	cycle	Cycle counter for RDCYCLE Instruction	Zicntr ~> 2.0.0

Address	Mode	Name	Long Name	Primary Extension
0xc01	U	time	Timer for RDTIME Instruction	Zicntr ~> 2.0.0
0xc02	U	instret	Instructions retired counter for RDINSTRET Instruction	Zicntr ~> 2.0.0
0xc03	U	hpmcounter3	User-mode Hardware Performance Counter 0	Zihpm ~> 2.0.0
0xc04	U	hpmcounter4	User-mode Hardware Performance Counter 1	Zihpm ~> 2.0.0
0xc05	U	hpmcounter5	User-mode Hardware Performance Counter 2	Zihpm ~> 2.0.0
0xc06	U	hpmcounter6	User-mode Hardware Performance Counter 3	Zihpm ~> 2.0.0
0xc07	U	hpmcounter7	User-mode Hardware Performance Counter 4	Zihpm ~> 2.0.0
0xc08	U	hpmcounter8	User-mode Hardware Performance Counter 5	Zihpm ~> 2.0.0
0xc09	U	hpmcounter9	User-mode Hardware Performance Counter 6	Zihpm ~> 2.0.0
0xc0a	U	hpmcounter10	User-mode Hardware Performance Counter 7	Zihpm ~> 2.0.0
0xc0b	U	hpmcounter11	User-mode Hardware Performance Counter 8	Zihpm ~> 2.0.0
0xc0c	U	hpmcounter12	User-mode Hardware Performance Counter 9	Zihpm ~> 2.0.0
0xc0d	U	hpmcounter13	User-mode Hardware Performance Counter 10	Zihpm ~> 2.0.0
0xc0e	U	hpmcounter14	User-mode Hardware Performance Counter 11	Zihpm ~> 2.0.0
0xc0f	U	hpmcounter15	User-mode Hardware Performance Counter 12	Zihpm ~> 2.0.0
0xc10	U	hpmcounter16	User-mode Hardware Performance Counter 13	Zihpm ~> 2.0.0
0xc11	U	hpmcounter17	User-mode Hardware Performance Counter 14	Zihpm ~> 2.0.0
0xc12	U	hpmcounter18	User-mode Hardware Performance Counter 15	Zihpm ~> 2.0.0
0xc13	U	hpmcounter19	User-mode Hardware Performance Counter 16	Zihpm ~> 2.0.0
0xc14	U	hpmcounter20	User-mode Hardware Performance Counter 17	Zihpm ~> 2.0.0
0xc15	U	hpmcounter21	User-mode Hardware Performance Counter 18	Zihpm ~> 2.0.0
0xc16	U	hpmcounter22	User-mode Hardware Performance Counter 19	Zihpm ~> 2.0.0
0xc17	U	hpmcounter23	User-mode Hardware Performance Counter 20	Zihpm ~> 2.0.0
0xc18	U	hpmcounter24	User-mode Hardware Performance Counter 21	Zihpm ~> 2.0.0
0xc19	U	hpmcounter25	User-mode Hardware Performance Counter 22	Zihpm ~> 2.0.0
0xc1a	U	hpmcounter26	User-mode Hardware Performance Counter 23	Zihpm ~> 2.0.0
0xc1b	U	hpmcounter27	User-mode Hardware Performance Counter 24	Zihpm ~> 2.0.0
0xc1c	U	hpmcounter28	User-mode Hardware Performance Counter 25	Zihpm ~> 2.0.0
0xc1d	U	hpmcounter29	User-mode Hardware Performance Counter 26	Zihpm ~> 2.0.0
0xc1e	U	hpmcounter30	User-mode Hardware Performance Counter 27	Zihpm ~> 2.0.0
0xc1f	U	hpmcounter31	User-mode Hardware Performance Counter 28	Zihpm ~> 2.0.0
0xc80	U	cycleh	High-half cycle counter for RDCYCLE Instruction	Zicntr ~> 2.0.0
0xc81	U	timeh	High-half timer for RDTIME Instruction	Zicntr ~> 2.0.0
0xc82	U	instreth	Instructions retired counter, high bits	Zicntr ~> 2.0.0

Appendix A: Extension Details

A.1. Extension A

Long Name: Atomic instructions
Version Requirement: = 2.1

2.1.0

State

ratified

Ratification date

2019-12

Implies

- Zaamo version 1.0.0
- Zalrsc version 1.0.0

A.1.1. Synopsis

The atomic-instruction extension, named A, contains instructions that atomically read-modify-write memory to support synchronization between multiple RISC-V harts running in the same memory space. The two forms of atomic instruction provided are load-reserved/store-conditional instructions and atomic fetch-and-op memory instructions. Both types of atomic instruction support various memory consistency orderings including unordered, acquire, release, and sequentially consistent semantics. These instructions allow RISC-V to support the RCsc memory consistency model. cite:[Gharachorloo90memoryconsistency]

After much debate, the language community and architecture community appear to have finally settled on release consistency as the standard memory consistency model and so the RISC-V atomic support is built around this model.

The A extension comprises instructions provided by the Zaamo and Zalrsc extensions.

A.1.2. Specifying Ordering of Atomic Instructions

The base RISC-V ISA has a relaxed memory model, with the `FENCE` instruction used to impose additional ordering constraints. The address space is divided by the execution environment into memory and I/O domains, and the `FENCE` instruction provides options to order accesses to one or both of these two address domains.

To provide more efficient support for release consistency cite:[Gharachorloo90memoryconsistency], each atomic instruction has two bits, *aq* and *rl*, used to specify additional memory ordering constraints as viewed by other RISC-V harts. The bits order accesses to one of the two address domains, memory or I/O, depending on which address domain the atomic instruction is accessing. No ordering constraint is implied to accesses to the other domain, and a FENCE instruction should be used to order across both domains.

If both bits are clear, no additional ordering constraints are imposed on the atomic memory operation. If only the *aq* bit is set, the atomic memory operation is treated as an *acquire* access, i.e., no following memory operations on this RISC-V hart can be observed to take place before the acquire memory operation. If only the *rl* bit is set, the atomic memory operation is treated as a *release* access, i.e., the release memory operation cannot be observed to take place before any earlier memory operations on this RISC-V hart. If both the *aq* and *rl* bits are set, the atomic memory operation is *sequentially consistent* and cannot be observed to happen before any earlier memory operations or after any later memory operations in the same RISC-V hart and to the same address domain.

A.1.3. Instructions

The following 76 instructions are added by extension version 2.1.0 (the minimum version of this extension that satifies the extension requirement).

amoadd.d.aq	**Atomic fetch-and-add doubleword (acquire)**
amoadd.d.aqrl	**Atomic fetch-and-add doubleword (acquire-release)**
amoadd.d.rl	**Atomic fetch-and-add doubleword (release)**
amoadd.d	**Atomic fetch-and-add doubleword**
amoadd.w.aq	**Atomic fetch-and-add word (acquire)**
amoadd.w.aqrl	**Atomic fetch-and-add word (acquire-release)**
amoadd.w.rl	**Atomic fetch-and-add word (release)**
amoadd.w	**Atomic fetch-and-add word**
amoand.d.aq	**Atomic fetch-and-and doubleword (acquire)**
amoand.d.aqrl	**Atomic fetch-and-and doubleword (acquire-release)**
amoand.d.rl	**Atomic fetch-and-and doubleword (release)**
amoand.d	**Atomic fetch-and-and doubleword**

amoand.w.aq	**Atomic fetch-and-and word (acquire)**
amoand.w.aqrl	**Atomic fetch-and-and word (acquire-release)**
amoand.w.rl	**Atomic fetch-and-and word (release)**
amoand.w	**Atomic fetch-and-and word**
amomax.d.aq	**Atomic MAX doubleword (acquire)**
amomax.d.aqrl	**Atomic MAX doubleword (acquire-release)**
amomax.d.rl	**Atomic MAX doubleword (release)**
amomax.d	**Atomic MAX doubleword**
amomax.w.aq	**Atomic MAX word (acquire)**
amomax.w.aqrl	**Atomic MAX word (acquire-release)**
amomax.w.rl	**Atomic MAX word (release)**
amomax.w	**Atomic MAX word**
amomaxu.d.aq	**Atomic MAX unsigned doubleword (acquire)**
amomaxu.d.aqrl	**Atomic MAX unsigned doubleword (acquire-release)**
amomaxu.d.rl	**Atomic MAX unsigned doubleword (release)**
amomaxu.d	**Atomic MAX unsigned doubleword**
amomaxu.w.aq	**Atomic MAX unsigned word (acquire)**
amomaxu.w.aqrl	**Atomic MAX unsigned word (acquire-release)**
amomaxu.w.rl	**Atomic MAX unsigned word (release)**
amomaxu.w	**Atomic MAX unsigned word**
amomin.d.aq	**Atomic MIN doubleword (acquire)**
amomin.d.aqrl	**Atomic MIN doubleword (acquire-release)**
amomin.d.rl	**Atomic MIN doubleword (release)**
amomin.d	**Atomic MIN doubleword**
amomin.w.aq	**Atomic MIN word (acquire)**
amomin.w.aqrl	**Atomic MIN word (acquire-release)**
amomin.w.rl	**Atomic MIN word (release)**
amomin.w	**Atomic MIN word**
amominu.d.aq	**Atomic MIN unsigned doubleword (acquire)**
amominu.d.aqrl	**Atomic MIN unsigned doubleword (acquire-release)**
amominu.d.rl	**Atomic MIN unsigned doubleword (release)**
amominu.d	**Atomic MIN unsigned doubleword**
amominu.w.aq	**Atomic MIN unsigned word (acquire)**
amominu.w.aqrl	**Atomic MIN unsigned word (acquire-release)**
amominu.w.rl	**Atomic MIN unsigned word (release)**
amominu.w	**Atomic MIN unsigned word**
amoor.d.aq	**Atomic fetch-and-or doubleword (acquire)**
amoor.d.aqrl	**Atomic fetch-and-or doubleword (acquire-release)**
amoor.d.rl	**Atomic fetch-and-or doubleword (release)**
amoor.d	**Atomic fetch-and-or doubleword**
amoor.w.aq	**Atomic fetch-and-or word (acquire)**
amoor.w.aqrl	**Atomic fetch-and-or word (acquire-release)**
amoor.w.rl	**Atomic fetch-and-or word (release)**
amoor.w	**Atomic fetch-and-or word**
amoswap.d.aq	**Atomic SWAP doubleword (acquire)**
amoswap.d.aqrl	**Atomic SWAP doubleword (acquire-release)**
amoswap.d.rl	**Atomic SWAP doubleword (release)**
amoswap.d	**Atomic SWAP doubleword**
amoswap.w.aq	**Atomic SWAP word (acquire)**
amoswap.w.aqrl	**Atomic SWAP word (acquire-release)**

amoswap.w.rl	**Atomic SWAP word (release)**
amoswap.w	**Atomic SWAP word**
amoxor.d.aq	**Atomic fetch-and-xor doubleword (acquire)**
amoxor.d.aqrl	**Atomic fetch-and-xor doubleword (acquire-release)**
amoxor.d.rl	**Atomic fetch-and-xor doubleword (release)**
amoxor.d	**Atomic fetch-and-xor doubleword**
amoxor.w.aq	**Atomic fetch-and-xor word (acquire)**
amoxor.w.aqrl	**Atomic fetch-and-xor word (acquire-release)**
amoxor.w.rl	**Atomic fetch-and-xor word (release)**
amoxor.w	**Atomic fetch-and-xor word**
lr.d	**Load reserved doubleword**
lr.w	**Load reserved word**
sc.d	**Store conditional doubleword**
sc.w	**Store conditional word**

A.1.4. OUT-OF-SCOPE Parameters

LRSC_FAIL_ON_NON_EXACT_LRSC ⇒ boolean

Whether or not a Store Conditional fails if its physical address and size do not exactly match the physical address and size of the last Load Reserved in program order (independent of whether or not the SC is in the current reservation set)

LRSC_FAIL_ON_VA_SYNONYM ⇒ boolean

Whether or not an `sc.l`/sc.d will fail if its VA does not match the VA of the prior `lr.l`/lr.d, even if the physical address of the SC and LR are the same

LRSC_MISALIGNED_BEHAVIOR ⇒ [always raise misaligned exception, always raise access fault, custom]

What to do when an LR/SC address is misaligned and MISALIGNED_AMO == false.

- 'always raise misaligned exception': self-explainitory
- 'always raise access fault': self-explainitory
- 'custom': Custom behavior; misaligned LR/SC may sometimes raise a misaligned exception and sometimes raise a access fault. Will lead to an 'unpredictable' call on any misaligned LR/SC access

LRSC_RESERVATION_STRATEGY ⇒ [reserve naturally-aligned 64-byte region, reserve naturally-aligned 128-byte region, reserve exactly enough to cover the access, custom]

Strategy used to handle reservation sets.

- "reserve naturally-aligned 64-byte region": Always reserve the 64-byte block containing the LR/SC address
- "reserve naturally-aligned 128-byte region": Always reserve the 128-byte block containing the LR/SC address
- "reserve exactly enough to cover the access": Always reserve exactly the LR/SC access, and no more
- "custom": Custom behavior, leading to an 'unpredictable' call on any LR/SC

MISALIGNED_AMO ⇒ boolean

whether or not the implementation supports misaligned atomics in main memory

MUTABLE_MISA_A ⇒ boolean

When the A extensions is supported, indicates whether or not the extension can be disabled in the misa.A bit.

A.2. Extension C

Long Name: Compressed instructions
Version Requirement: = 2.0

2.0.0

State

ratified

Ratification date

2019-12

Implies

- Zca version 1.0.0

- Zcf version 1.0.0
- Zcd version 1.0.0

A.2.1. Synopsis

The C extension reduces static and dynamic code size by adding short 16-bit instruction encodings for common operations. The C extension can be added to any of the base ISAs (RV32, RV64, RV128), and we use the generic term "RVC" to cover any of these. Typically, 50%-60% of the RISC-V instructions in a program can be replaced with RVC instructions, resulting in a 25%-30% code-size reduction.

A.2.2. Overview

RVC uses a simple compression scheme that offers shorter 16-bit versions of common 32-bit RISC-V instructions when:

- the immediate or address offset is small, or
- one of the registers is the zero register (`x0`), the ABI link register (`x1`), or the ABI stack pointer (`x2`), or
- the destination register and the first source register are identical, or
- the registers used are the 8 most popular ones.

The C extension is compatible with all other standard instruction extensions. The C extension allows 16-bit instructions to be freely intermixed with 32-bit instructions, with the latter now able to start on any 16-bit boundary, i.e., IALIGN=16. With the addition of the C extension, no instructions can raise instruction-address-misaligned exceptions.

Removing the 32-bit alignment constraint on the original 32-bit instructions allows significantly greater code density.

The compressed instruction encodings are mostly common across RV32C, RV64C, and RV128C, but as shown in Table 34, a few opcodes are used for different purposes depending on base ISA. For example, the wider address-space RV64C and RV128C variants require additional opcodes to compress loads and stores of 64-bit integer values, while RV32C uses the same opcodes to compress loads and stores of single-precision floating-point values. Similarly, RV128C requires additional opcodes to capture loads and stores of 128-bit integer values, while these same opcodes are used for loads and stores of double-precision floating-point values in RV32C and RV64C. If the C extension is implemented, the appropriate compressed floating-point load and store instructions must be provided whenever the relevant standard floating-point extension (F and/or D) is also implemented. In addition, RV32C includes a compressed jump and link instruction to compress short-range subroutine calls, where the same opcode is used to compress ADDIW for RV64C and RV128C.

Double-precision loads and stores are a significant fraction of static and dynamic instructions, hence the motivation to include them in the RV32C and RV64C encoding.

Although single-precision loads and stores are not a significant source of static or dynamic compression for benchmarks compiled for the currently supported ABIs, for microcontrollers that only provide hardware single-precision floating-point units and have an ABI that only supports single-precision floating-point numbers, the single-precision loads and stores will be used at least as frequently as double-precision loads and stores in the measured benchmarks. Hence, the motivation to provide compressed support for these in RV32C.

Short-range subroutine calls are more likely in small binaries for microcontrollers, hence the motivation to include these in RV32C.

Although reusing opcodes for different purposes for different base ISAs adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISAs. The compressed floating-point load and store variants use the same instruction format with the same register specifiers as the wider integer loads and stores.

RVC was designed under the constraint that each RVC instruction expands into a single 32-bit instruction in either the base ISA (RV32I/E, RV64I/E, or RV128I) or the F and D standard extensions where present. Adopting this constraint has two main benefits:

- Hardware designs can simply expand RVC instructions during decode, simplifying verification and minimizing modifications to existing microarchitectures.
- Compilers can be unaware of the RVC extension and leave code compression to the assembler and linker, although a compression-aware compiler will generally be able to produce better results.

We felt the multiple complexity reductions of a simple one-one mapping between C and base IFD instructions far outweighed the potential gains of a slightly denser encoding that added additional instructions only supported in the C extension, or that allowed encoding of multiple IFD instructions in one C instruction.

It is important to note that the C extension is not designed to be a stand-alone ISA, and is meant to be used alongside a base ISA.

Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch cite:[stretch], developed in the late 1950s, had an ISA with 32-bit and 64-bit instructions, where some of the 32-bit instructions were compressed versions of the full 64-bit instructions. Stretch also employed the concept of limiting the set of registers that were addressable in some of the shorter instruction formats, with short branch instructions that could only refer to one of the index registers. The later IBM 360 architecture cite:[ibm360] supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.

In 1963, CDC introduced the Cray-designed CDC 6600 cite:[cdc6600], a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction

format, with 16-bit and 32-bit instruction lengths.

The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced code size relative to their starting points by about 25-30%, yielding code that was significantly smaller than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.

Since the original RISC ISAs did not leave sufficient opcode space free to include these unplanned compressed instructions, they were instead developed as complete new ISAs. This meant compilers needed different code generators for the separate compressed ISAs. The first compressed RISC ISA extensions (e.g., ARM Thumb and MIPS16) used only a fixed 16-bit instruction size, which gave good reductions in static code size but caused an increase in dynamic instruction count, which led to lower performance compared to the original fixed-width 32-bit instruction size. This led to the development of a second generation of compressed RISC ISA designs with mixed 16-bit and 32-bit instruction lengths (e.g., ARM Thumb2, microMIPS, PowerPC VLE), so that performance was similar to pure 32-bit instructions but with significant code size savings. Unfortunately, these different generations of compressed ISAs are incompatible with each other and with the original uncompressed ISA, leading to significant complexity in documentation, implementations, and software tools support.

Of the commonly used 64-bit ISAs, only PowerPC and microMIPS currently supports a compressed instruction format. It is surprising that the most popular 64-bit ISA for mobile platforms (ARM v8) does not include a compressed instruction format given that static code size and dynamic instruction fetch bandwidth are important metrics. Although static code size is not a major concern in larger systems, instruction fetch bandwidth can be a major bottleneck in servers running commercial workloads, which often have a large instruction working set.

> Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications *and* to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size. cite:[waterman-ms]

A.2.3. Compressed Instruction Formats

Table 4 shows the nine compressed instruction formats. CR, CI, and CSS can use any of the 32 RVI registers, but CIW, CL, CS, CA, and CB are limited to just 8 of them. Table 5 lists these popular registers, which correspond to registers `x8` to `x15`. Note that there is a separate version of load and store instructions that use the stack pointer as the base address register, since saving to and restoring from the stack are so prevalent, and that they use the CI and CSS formats to allow access to all 32 data registers. CIW supplies an 8-bit immediate for the ADDI4SPN instruction.

The RISC-V ABI was changed to make the frequently used registers map to registers 'x8-x15'. This simplifies the decompression decoder by having a contiguous naturally aligned set of register numbers, and is also compatible with the RV32E and RV64E base ISAs, which only have 16 integer registers.

Compressed register-based floating-point loads and stores also use the CL and CS formats respectively, with the eight registers mapping to `f8` to `f15`.

The standard RISC-V calling convention maps the most frequently used floating-point registers to registers `f8` to `f15`, which allows the same register decompression decoding as for integer register numbers.

The formats were designed to keep bits for the two register source specifiers in the same place in all instructions, while the destination register field can move. When the full 5-bit destination register specifier is present, it is in the same place as in the 32-bit RISC-V encoding. Where immediates are sign-extended, the sign extension is always from bit 12. Immediate fields have been scrambled, as in the base specification, to reduce the number of immediate muxes required.

The immediate fields are scrambled in the instruction formats instead of in sequential order so that as many bits as possible are in the same position in every instruction, thereby simplifying implementations.

For many RVC instructions, zero-valued immediates are disallowed and `x0` is not a valid 5-bit register specifier. These restrictions free up encoding space for other instructions requiring fewer operand bits.

Table 4. Compressed 16-bit RVC instruction formats

Format	Meaning	15 14 13 12		11 10 9 8 7		6 5 4 3 2		1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd'	op
CL	Load	funct3	imm		rs1'	imm	rd'	op
CS	Store	funct3	imm		rs1'	imm	rs2'	op
CA	Arithmetic	funct6			rd'/rs1'	funct2	rs2'	op
CB	Branch/Arithmetic	funct3	offset		rd'/rs1'	offset		op
CJ	Jump	funct3	jump target					op

Table 5. Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, CA, and CB formats.

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

A.2.4. Instructions

The following 8 instructions are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

c.fld	**Load double-precision**
c.fldsp	**Load doubleword into floating-point register from stack**
c.fsd	**Store double-precision**
c.fsdsp	**Store double-precision value to stack**
c.flw	**Load single-precision**
c.flwsp	**Load word into floating-point register from stack**
c.fsw	**Store single-precision**
c.fswsp	**Store single-precision value to stack**

A.2.5. OUT-OF-SCOPE Parameters

MUTABLE_MISA_C ⇒ boolean

Indicates whether or not the C extension can be disabled with the misa.C bit.

A.3. Extension D

Long Name: Double-precision floating-point
Version Requirement: = 2.2

2.2.0

State

ratified

Ratification date

2019-12

Changes

- Define NaN-boxing scheme, changed definition of FMAX and FMIN

Implies

- F version 2.2.0

A.3.1. Synopsis

The D extension adds double-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard. The D extension depends on the base single-precision instruction subset F.

A.3.2. D Register State

The D extension widens the 32 floating-point registers, `f0-f31`, to 64 bits (FLEN=64 in Table 6. The `f` registers can now hold either 32-bit or 64-bit floating-point values as described below in Section A.3.3.

FLEN can be 32, 64, or 128 depending on which of the F, D, and Q extensions are supported. There can be up to four different floating-point precisions supported, including H, F, D, and Q.

A.3.3. NaN Boxing of Narrower Values

When multiple floating-point precisions are supported, then valid values of narrower *n*-bit types, *n*<FLEN, are represented in the lower *n* bits of an FLEN-bit NaN value, in a process termed NaN-boxing. The upper bits of a valid NaN-boxed value must be all 1s. Valid NaN-boxed *n*-bit values therefore appear as negative quiet NaNs (qNaNs) when viewed as any wider *m*-bit value, $n < m \leq$ FLEN. Any operation that writes a narrower result to an 'f' register must write all 1s to the uppermost FLEN-*n* bits to yield a legal NaN-boxedvalue.

Software might not know the current type of data stored in a floating-point register but has to be able to save and restore the register values, hence the result of using wider operations to transfer narrower values has to be defined. A common case is for callee-saved registers, but a standard convention is also desirable for features including varargs, user-level threading libraries, virtual machine migration, and debugging.

Floating-point *n*-bit transfer operations move external values held in IEEE standard formats into and out of the `f` registers, and comprise floating-point loads and stores (FL*n*/FS*n*) and floating-point move instructions (FMV.*n*.X/FMV.X.*n*). A narrower *n*-bit transfer, *n*<FLEN, into the `f` registers will create a valid NaN-boxed value. A narrower *n*-bit transfer out of the floating-point registers will transfer the lower *n* bits of the register ignoring the upper FLEN-*n* bits.

Apart from transfer operations described in the previous paragraph, all other floating-point operations on narrower *n*-bit operations, *n*<FLEN, check if the input operands are correctly NaN-boxed, i.e., all upper FLEN-*n* bits are 1. If so, the *n* least-significant bits of the input are used as the input value, otherwise the input value is treated as an *n*-bit canonical NaN.

Earlier versions of this document did not define the behavior of feeding the results of narrower or wider operands into an operation, except to require that wider saves and restores would preserve the value of a narrower operand. The new definition removes this implementation-specific behavior, while still accommodating both non-recoded and recoded implementations of the floating-point unit. The new definition also helps catch software errors by propagating NaNs if values are used incorrectly.

Non-recoded implementations unpack and pack the operands to IEEE standard format on the input and output of every floating-point operation. The NaN-boxing cost to a non-recoded implementation is primarily in checking if the upper bits of a narrower operation represent a legal NaN-boxed value, and in writing all 1s to the upper bits of a result.

Recoded implementations use a more convenient internal format to represent floating-point values, with an added exponent bit to allow all values to be held normalized. The cost to the recoded implementation is primarily the extra tagging needed to track the internal types and sign bits, but this can be done without adding new state bits by recoding NaNs internally in the exponent field. Small modifications are needed to the pipelines used to transfer values in and out of the recoded format, but the datapath and latency costs are minimal. The recoding process has to handle shifting of input subnormal values for wide operands in any case, and extracting the NaN-boxed value is a similar process to normalization except for skipping over leading-1 bits instead of skipping over leading-0 bits, allowing the datapath muxing to be shared.

The rationale to not include Q as a profile option is that quad-precision floating-point is unlikely to be implemented in hardware, and so we do not require or expect software to expend effort optimizing use of Q instructions in case they are present.

A.3.4. Instructions

The following 82 instructions are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

fadd.d	**Floating-Point Add Double-Precision**
fclass.d	**Floating-Point Classify Double-Precision**
fcvt.d.l	**Floating-Point Convert Long to Double-Precision**
fcvt.d.lu	**Floating-Point Convert Unsigned Long to Double-Precision**
fcvt.d.s	**Floating-Point Convert Single-Precision to Double-Precision**
fcvt.d.w	**Floating-Point Convert Word to Double-Precision**
fcvt.d.wu	**Floating-Point Convert Unsigned Word to Double-Precision**
fcvt.l.d	**Floating-Point Convert Double-Precision to Long**
fcvt.lu.d	**Floating-Point Convert Double-Precision to Unsigned Long**
fcvt.s.d	**Floating-Point Convert Double-Precision to Single-Precision**

fcvt.w.d	**Floating-Point Convert Double-Precision to Word**
fcvt.wu.d	**Floating-Point Convert Double-Precision to Unsigned Word**
fcvtmod.w.d	**Floating-Point Convert Double-Precision to Word with Modulo**
fdiv.d	**Floating-Point Divide Double-Precision**
feq.d	**Floating-Point Equal Double-Precision**
fld	**Floating-Point Load Double-Precision**
fle.d	**Floating-Point Less Than or Equal Double-Precision**
fleq.d	**Floating-Point Less Than or Equal Quiet Double-Precision**
fli.d	**Floating-Point Load Immediate Double-Precision**
flt.d	**Floating-Point Less Than Double-Precision**
fltq.d	**Floating-Point Less Than Quiet Double-Precision**
fmadd.d	**Floating-Point Multiply-Add Double-Precision**
fmax.d	**Floating-Point Maximum-Number Double-Precision**
fmaxm.d	**Floating-Point Maximum-Number NaN Double-Precision**
fmin.d	**Floating-Point Minimum-Number Double-Precision**
fminm.d	**Floating-Point Minimum-Number NaN Double-Precision**
fmsub.d	**Floating-Point Multiply-Subtract Double-Precision**
fmul.d	**Floating-Point Multiply Double-Precision**
fmv.d.x	**Floating-Point Move from Integer Register to Double-Precision Register**
fmv.x.d	**Floating-Point Move from Double-Precision Register to Integer Register**
fmvh.x.d	**Floating-Point Move High Half from Double-Precision Register to Integer Register**
fmvp.d.x	**Floating-Point Move Pair from Integer Registers to Double-Precision Register**
fnmadd.d	**Floating-Point Negate-Multiply-Add Double-Precision**
fnmsub.d	**Floating-Point Negate-Multiply-Subtract Double-Precision**
fround.d	**Floating-Point Round Double-Precision**
froundnx.d	**Floating-Point Round-to-Integer Inexact Double-Precision**
fsd	**Floating-Point Store Double-Precision**
fsgnj.d	**Floating-Point Sign-Inject Double-Precision**
fsgnjn.d	**Floating-Point Sign-Inject Negate Double-Precision**
fsgnjx.d	**Floating-Point Sign-Inject XOR Double-Precision**
fsqrt.d	**Floating-Point Square Root Double-Precision**
fsub.d	**Floating-Point Subtract Double-Precision**
fadd.s	**Floating-Point Add Single-Precision**
fclass.s	**Floating-Point Classify Single-Precision**
fcvt.l.s	**Floating-Point Convert Single-Precision to Long**
fcvt.lu.s	**Floating-Point Convert Single-Precision to Unsigned Long**
fcvt.s.l	**Floating-Point Convert Long to Single-Precision**
fcvt.s.lu	**Floating-Point Convert Unsigned Long to Single-Precision**
fcvt.s.w	**Floating-Point Convert Word to Single-Precision**
fcvt.s.wu	**Floating-Point Convert Unsigned Word to Single-Precision**
fcvt.w.s	**Floating-Point Convert Single-Precision to Word**
fcvt.wu.s	**Floating-Point Convert Single-Precision to Unsigned Word**
fdiv.s	**Floating-Point Divide Single-Precision**
feq.s	**Floating-Point Equal Single-Precision**
fle.s	**Floating-Point Less Than or Equal Single-Precision**
flt.s	**Floating-Point Less Than Single-Precision**
flw	**Floating-Point Load Single-Precision**
fmadd.s	**Floating-Point Multiply-Add Single-Precision**
fmax.s	**Floating-Point Maximum-Number Single-Precision**
fmin.s	**Floating-Point Minimum-Number Single-Precision**

fmsub.s	**Floating-Point Multiply-Subtract Single-Precision**
fmul.s	**Floating-Point Multiply Single-Precision**
fmv.w.x	**Floating-Point Move Single-Precision Word from Integer Register**
fmv.x.w	**Floating-Point Move Single-Precision Word to Integer Register**
fnmadd.s	**Floating-Point Negate-Multiply-Add Single-Precision**
fnmsub.s	**Floating-Point Negate-Multiply-Subtract Single-Precision**
fsgnj.s	**Floating-Point Sign-Inject Single-Precision**
fsgnjn.s	**Floating-Point Sign-Inject Negate Single-Precision**
fsgnjx.s	**Floating-Point Sign-Inject XOR Single-Precision**
fsqrt.s	**Floating-Point Square Root Single-Precision**
fsub.s	**Floating-Point Subtract Single-Precision**
fsw	**Floating-Point Store Single-Precision**
c.fld	**Load double-precision**
c.fldsp	**Load doubleword into floating-point register from stack**
c.fsd	**Store double-precision**
c.fsdsp	**Store double-precision value to stack**
c.flw	**Load single-precision**
c.flwsp	**Load word into floating-point register from stack**
c.fsw	**Store single-precision**
c.fswsp	**Store single-precision value to stack**
fcvt.d.h	**No synopsis available**
fcvt.h.d	**Floating-point Convert Double-precision to Half-precision**

A.3.5. CSRs

The following 3 CSRs are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
fcsr	Floating-point control and status register (frm + fflags)	0x3	U
fflags	Floating-Point Accrued Exceptions	0x1	U
frm	Floating-Point Dynamic Rounding Mode	0x2	U

A.3.6. OUT-OF-SCOPE Parameters

MUTABLE_MISA_D ⇒ boolean

Indicates whether or not the D extension can be disabled with the misa.D bit.

A.4. Extension F

Long Name: Single-precision floating-point
Version Requirement: = 2.2

2.2.0

State

ratified

Ratification date

2019-12

Changes

- Define NaN-boxing scheme, changed definition of FMAX and FMIN

A.4.1. Synopsis

This chapter describes the standard instruction-set extension for single-precision floating-point, which is named "F" and adds single-precision floating-point computational instructions compliant with the IEEE 754-2008 arithmetic standard cite:[ieee754-2008]. The F extension depends on the "Zicsr" extension for control and status register access.

A.4.2. F Register State

The F extension adds 32 floating-point registers, `f0-f31`, each 32 bits wide, and a floating-point control and status register fcsr, which contains the operating mode and exception status of the floating-point unit. This additional state is shown in Table 6. We use the term FLEN to describe the width of the floating-point registers in the RISC-V ISA, and FLEN=32 for the F single-precision floating-point extension. Most floating-point instructions operate on values in the floating-point register file. Floating-point load and store instructions transfer floating-point values between registers and memory. Instructions to transfer values to and from the integer register file are also provided.

We considered a unified register file for both integer and floating-point values as this simplifies software register allocation and calling conventions, and reduces total user state. However, a split organization increases the total number of registers accessible with a given instruction width, simplifies provision of enough regfile ports for wide superscalar issue, supports decoupled floating-point-unit architectures, and simplifies use of internal floating-point encoding techniques. Compiler support and calling conventions for split register file architectures are well understood, and using dirty bits on floating-point register file state can reduce context-switch overhead.

Table 6. RISC-V standard F extension single-precision floating-point state

FLEN-1	0
f0	
f1	
f2	
f3	
f4	
f5	
f6	
f7	
f8	
f9	
f10	
f11	
f12	
f13	
f14	
f15	
f16	
f17	
f18	
f19	
f20	
f21	
f22	
f23	
f24	
f25	
f26	
f27	
f28	
f29	
f30	
f31	
FLEN	
31	0
fcsr	
32	

Floating-Point Control and Status Register

The floating-point control and status register, fcsr, is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in Floating-Point Control and Status Register.

Floating-point control and status register

Unresolved directive in RVI20-64-CRD.adoc - include::images/wavedrom/float-csr.adoc[]

The fcsr register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads fcsr by copying it into integer register *rd*. FSCSR swaps the value in fcsr by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into fcsr.

The fields within the fcsr can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field frm (fcsr bits 7—5) and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in frm by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into frm. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field fflags (fcsr bits 4—0).

Bits 31—8 of the fcsr are reserved for other standard extensions. If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in frm. Rounding modes are encoded as shown in Table 7. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in frm. The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111). Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

Table 7. Rounding mode encoding.

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards `-\infty`)
011	RUP	Round Up (towards `+\infty`)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Reserved for future use.*
110		*Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *reserved.*

The C99 language standard effectively mandates the provision of a dynamic rounding mode register. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline. Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.

The ratified version of the F spec mandated that an illegal-instruction exception was raised when an instruction was executed with a reserved dynamic rounding mode. This has been weakened to reserved, which matches the behavior of static rounding-mode instructions. Raising an illegal-instruction exception is still valid behavior when encountering a reserved encoding, so implementations compatible with the ratified spec are compatible with the weakened spec.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 8. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 8. Accrued exception flag encoding.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

A.4.3. NaN Generation and Propagation

Except when otherwise stated, if the result of a floating-point operation is NaN, it is the canonical NaN. The canonical NaN has a positive sign and all significand bits clear except the MSB, a.k.a. the quiet bit. For single-precision floating-point, this corresponds to the pattern `0x7fc00000`.

We considered propagating NaN payloads, as is recommended by the standard, but this decision would have increased hardware cost. Moreover, since this feature is optional in the standard, it cannot be used in portable code.

Implementers are free to provide a NaN payload propagation scheme as a nonstandard extension enabled by a nonstandard

operating mode. However, the canonical NaN scheme described above must always be supported and should be the default mode.

We require implementations to return the standard-mandated default values in the case of exceptional conditions, without any further intervention on the part of user-level software (unlike the Alpha ISA floating-point trap barriers). We believe full hardware handling of exceptional cases will become more common, and so wish to avoid complicating the user-level ISA to optimize other approaches. Implementations can always trap to machine-mode software handlers to provide exceptional default values.

A.4.4. Subnormal Arithmetic

Operations on subnormal numbers are handled in accordance with the IEEE 754-2008 standard.

In the parlance of the IEEE standard, tininess is detected after rounding.

Detecting tininess after rounding results in fewer spurious underflow signals.

A.4.5. Instructions

The following 34 instructions are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

fadd.s	**Floating-Point Add Single-Precision**
fclass.s	**Floating-Point Classify Single-Precision**
fcvt.l.s	**Floating-Point Convert Single-Precision to Long**
fcvt.lu.s	**Floating-Point Convert Single-Precision to Unsigned Long**
fcvt.s.l	**Floating-Point Convert Long to Single-Precision**
fcvt.s.lu	**Floating-Point Convert Unsigned Long to Single-Precision**
fcvt.s.w	**Floating-Point Convert Word to Single-Precision**
fcvt.s.wu	**Floating-Point Convert Unsigned Word to Single-Precision**
fcvt.w.s	**Floating-Point Convert Single-Precision to Word**
fcvt.wu.s	**Floating-Point Convert Single-Precision to Unsigned Word**
fdiv.s	**Floating-Point Divide Single-Precision**
feq.s	**Floating-Point Equal Single-Precision**
fle.s	**Floating-Point Less Than or Equal Single-Precision**
flt.s	**Floating-Point Less Than Single-Precision**
flw	**Floating-Point Load Single-Precision**
fmadd.s	**Floating-Point Multiply-Add Single-Precision**
fmax.s	**Floating-Point Maximum-Number Single-Precision**
fmin.s	**Floating-Point Minimum-Number Single-Precision**
fmsub.s	**Floating-Point Multiply-Subtract Single-Precision**
fmul.s	**Floating-Point Multiply Single-Precision**
fmv.w.x	**Floating-Point Move Single-Precision Word from Integer Register**
fmv.x.w	**Floating-Point Move Single-Precision Word to Integer Register**
fnmadd.s	**Floating-Point Negate-Multiply-Add Single-Precision**
fnmsub.s	**Floating-Point Negate-Multiply-Subtract Single-Precision**
fsgnj.s	**Floating-Point Sign-Inject Single-Precision**
fsgnjn.s	**Floating-Point Sign-Inject Negate Single-Precision**
fsgnjx.s	**Floating-Point Sign-Inject XOR Single-Precision**
fsqrt.s	**Floating-Point Square Root Single-Precision**
fsub.s	**Floating-Point Subtract Single-Precision**
fsw	**Floating-Point Store Single-Precision**
c.flw	**Load single-precision**
c.flwsp	**Load word into floating-point register from stack**
c.fsw	**Store single-precision**
c.fswsp	**Store single-precision value to stack**

A.4.6. CSRs

The following 3 CSRs are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
fcsr	Floating-point control and status register (frm + fflags)	0x3	U
fflags	Floating-Point Accrued Exceptions	0x1	U
frm	Floating-Point Dynamic Rounding Mode	0x2	U

A.4.7. OUT-OF-SCOPE Parameters

HW_MSTATUS_FS_DIRTY_UPDATE ⇒ [never, precise, imprecise]

Indicates whether or not hardware will write to mstatus.FS

Values are:

never	Hardware never writes mstatus.FS
precise	Hardware writes mstatus.FS to the Dirty (3) state precisely when F registers are modified
imprecise	Hardware writes mstatus.FS imprecisely. This will result in a call to unpredictable() on any attempt to read mstatus or write FP state.

MSTATUS_FS_LEGAL_VALUES ⇒ at most 4-element array of [0, 1, 2, 3]

The set of values that mstatus.FS will accept from a software write.

MUTABLE_MISA_F ⇒ boolean

Indicates whether or not the F extension can be disabled with the misa.F bit.

A.5. Extension I

Long Name: Base integer ISA (RV32I or RV64I)
Version Requirement: ~> 2.1

2.1.0

State

ratified

Ratification date

2019-06

Changes

- ratified RVWMO memory model and exclusion of FENCE.I, counters, and CSR instructions that were in previous base ISA

A.5.1. Synopsis

Base integer instructions — TODO

RVI is the mandatory base ISA for RVA, and is little-endian.

As per the unprivileged architecture specification, the ecall instruction causes a requested trap to the execution environment.

Misaligned loads and stores might not be supported.

The fence.tso instruction is mandatory.

The fence.tso instruction was incorrectly described as optional in the 2019 ratified specifications. However, fence.tso is encoded within the standard fence encoding such that implementations must treat it as a simple global fence if they do not natively support TSO-ordering optimizations. As software can always assume without any penalty that fence.tso is being exploited by a hardware implementation, there is no advantage to making the instruction a profile option. Later versions of the unprivileged ISA specifications correctly indicate that fence.tso is mandatory.

A.5.2. Instructions

The following 53 instructions are added by extension version 2.1.0 (the minimum version of this extension that satifies the extension requirement).

add	**Integer add**

addi	**Add immediate**
addiw	**Add immediate word**
addw	**Add word**
and	**And**
andi	**And immediate**
auipc	**Add upper immediate to pc**
beq	**Branch if equal**
bge	**Branch if greater than or equal**
bgeu	**Branch if greater than or equal unsigned**
blt	**Branch if less than**
bltu	**Branch if less than unsigned**
bne	**Branch if not equal**
ebreak	**Breakpoint exception**
ecall	**Environment call**
fence.tso	**Memory ordering fence, total store ordering**
fence	**Memory ordering fence**
jal	**Jump and link**
jalr	**Jump and link register**
lb	**Load byte**
lbu	**Load byte unsigned**
ld	**Load doubleword**
lh	**Load halfword**
lhu	**Load halfword unsigned**
lui	**Load upper immediate**
lw	**Load word**
lwu	**Load word unsigned**
or	**Or**
ori	**Or immediate**
sb	**Store byte**
sd	**Store doubleword**
sh	**Store halfword**
sll	**Shift left logical**
slli	**Shift left logical immediate**
slliw	**Shift left logical immediate word**
sllw	**Shift left logical word**
slt	**Set on less than**
slti	**Set on less than immediate**
sltiu	**Set on less than immediate unsigned**
sltu	**Set on less than unsigned**
sra	**Shift right arithmetic**
srai	**Shift right arithmetic immediate**
sraiw	**Shift right arithmetic immediate word**
sraw	**Shift right arithmetic word**
srl	**Shift right logical**
srli	**Shift right logical immediate**
srliw	**Shift right logical immediate word**
srlw	**Shift right logical word**
sub	**Subtract**
subw	**Subtract word**
sw	**Store word**

xor	**Exclusive Or**
xori	**Exclusive Or immediate**

A.6. Extension M

Long Name: Integer multiply and divide instructions
Version Requirement: = 2.0

2.0.0

State

ratified

Ratification date

2019-12

A.6.1. Synopsis

This chapter describes the standard integer multiplication and division instruction extension, which is named M and contains instructions that multiply or divide values held in two integer registers.

We separate integer multiply and divide out from the base to simplify low-end implementations, or for applications where integer multiply and divide operations are either infrequent or better handled in attached accelerators.

A.6.2. Instructions

The following 13 instructions are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

div	**Signed division**
divu	**Unsigned division**
divuw	**Unsigned 32-bit division**
divw	**Signed 32-bit division**
mul	**Signed multiply**
mulh	**Signed multiply high**
mulhsu	**Signed/unsigned multiply high**
mulhu	**Unsigned multiply high**
mulw	**Signed 32-bit multiply**
rem	**Signed remainder**
remu	**Unsigned remainder**
remuw	**Unsigned 32-bit remainder**
remw	**Signed 32-bit remainder**

A.6.3. OUT-OF-SCOPE Parameters

MUTABLE_MISA_M ⇒ boolean

Indicates whether or not the M extension can be disabled with the misa.M bit.

A.7. Extension Zicntr

Long Name: Architectural performance counters
Version Requirement: = 2.0

2.0.0

State

ratified

Ratification date

2019-12

A.7.1. Synopsis

Architectural performance counters

A.7.2. CSRs

The following 10 CSRs are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
cycle	Cycle counter for RDCYCLE Instruction	0xc00	U
cycleh	High-half cycle counter for RDCYCLE Instruction	0xc80	U
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U
instreth	Instructions retired counter, high bits	0xc82	U
mcycle	Machine Cycle Counter	0xb00	M
mcycleh	High-half machine Cycle Counter	0xb80	M
minstret	Machine Instructions Retired Counter	0xb02	M
minstreth	Machine Instructions Retired Counter	0xb82	M
time	Timer for RDTIME Instruction	0xc01	U
timeh	High-half timer for RDTIME Instruction	0xc81	U

A.7.3. OUT-OF-SCOPE Parameters

TIME_CSR_IMPLEMENTED ⇒ boolean

Whether or not a real hardware time CSR exists. Implementations can either provide a real CSR or emulate access at M-mode.

Possible values:

true

time/timeh exists, and accessing it will not cause an IllegalInstruction trap

false

time/timeh does not exist. Accessing the CSR will cause an IllegalInstruction trap or enter an unpredictable state, depending on TRAP_ON_UNIMPLEMENTED_CSR. Privileged software may emulate the time CSR, or may pass the exception to a lower level.

A.8. Extension Zifencei

Long Name: Instruction fence
Version Requirement: = 2.0

2.0.0

State

ratified

A.8.1. Synopsis

This chapter defines the "Zifencei" extension, which includes the FENCE.I instruction that provides explicit synchronization between writes to instruction memory and instruction fetches on the same hart. Currently, this instruction is the only standard mechanism to ensure that stores visible to a hart will also be visible to its instruction fetches.

We considered but did not include a "store instruction word" instruction as in cite:[majc]. JIT compilers may generate a large trace of instructions before a single FENCE.I, and amortize any instruction cache snooping/invalidation overhead by writing translated instructions to memory regions that are known not to reside in the I-cache.

The FENCE.I instruction was designed to support a wide variety of implementations. A simple implementation can flush the local instruction cache and the instruction pipeline when the FENCE.I is executed. A more complex implementation might snoop the instruction (data) cache on every data (instruction) cache miss, or use an inclusive unified private L2 cache to invalidate lines from the primary instruction cache when they are being written by a local store instruction. If instruction and data caches are kept coherent in this way, or if the memory system consists of only uncached RAMs, then just the fetch pipeline needs to be flushed at a FENCE.I.

The FENCE.I instruction was previously part of the base I instruction set. Two main issues are driving moving this out of the mandatory base, although at time of writing it is still the only standard method for maintaining instruction-fetch coherence.

First, it has been recognized that on some systems, FENCE.I will be expensive to implement and alternate mechanisms are being discussed in the memory model task group. In particular, for designs that have an incoherent instruction cache and an incoherent data cache, or where the instruction cache refill does not snoop a coherent data cache, both caches must be completely flushed when a FENCE.I instruction is encountered. This problem is exacerbated when there are multiple levels of I and D cache in front of a unified cache or outer memory system.

> Second, the instruction is not powerful enough to make available at user level in a Unix-like operating system environment. The FENCE.I only synchronizes the local hart, and the OS can reschedule the user hart to a different physical hart after the FENCE.I. This would require the OS to execute an additional FENCE.I as part of every context migration. For this reason, the standard Linux ABI has removed FENCE.I from user-level and now requires a system call to maintain instruction-fetch coherence, which allows the OS to minimize the number of FENCE.I executions required on current systems and provides forward-compatibility with future improved instruction-fetch coherence mechanisms.
>
> Future approaches to instruction-fetch coherence under discussion include providing more restricted versions of FENCE.I that only target a given address specified in *rs1*, and/or allowing software to use an ABI that relies on machine-mode cache-maintenance operations.

A.8.2. Instructions

The following 1 instructions are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

fence.i	**Instruction fence**

A.9. Extension Zihpm

Long Name: Programmable hardware performance counters
Version Requirement: = 2.0

2.0.0

State

ratified

A.9.1. Synopsis

Programmable hardware performance counters

The number of counters is platform-specific.

A.9.2. CSRs

The following 29 CSRs are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
hpmcounter10	User-mode Hardware Performance Counter 7	0xc0a	U
hpmcounter11	User-mode Hardware Performance Counter 8	0xc0b	U
hpmcounter12	User-mode Hardware Performance Counter 9	0xc0c	U
hpmcounter13	User-mode Hardware Performance Counter 10	0xc0d	U
hpmcounter14	User-mode Hardware Performance Counter 11	0xc0e	U
hpmcounter15	User-mode Hardware Performance Counter 12	0xc0f	U
hpmcounter16	User-mode Hardware Performance Counter 13	0xc10	U
hpmcounter17	User-mode Hardware Performance Counter 14	0xc11	U
hpmcounter18	User-mode Hardware Performance Counter 15	0xc12	U
hpmcounter19	User-mode Hardware Performance Counter 16	0xc13	U
hpmcounter20	User-mode Hardware Performance Counter 17	0xc14	U
hpmcounter21	User-mode Hardware Performance Counter 18	0xc15	U
hpmcounter22	User-mode Hardware Performance Counter 19	0xc16	U
hpmcounter23	User-mode Hardware Performance Counter 20	0xc17	U
hpmcounter24	User-mode Hardware Performance Counter 21	0xc18	U
hpmcounter25	User-mode Hardware Performance Counter 22	0xc19	U
hpmcounter26	User-mode Hardware Performance Counter 23	0xc1a	U
hpmcounter27	User-mode Hardware Performance Counter 24	0xc1b	U
hpmcounter28	User-mode Hardware Performance Counter 25	0xc1c	U
hpmcounter29	User-mode Hardware Performance Counter 26	0xc1d	U
hpmcounter3	User-mode Hardware Performance Counter 0	0xc03	U
hpmcounter30	User-mode Hardware Performance Counter 27	0xc1e	U
hpmcounter31	User-mode Hardware Performance Counter 28	0xc1f	U

Name	Long Name	Address	Mode
hpmcounter4	User-mode Hardware Performance Counter 1	0xc04	U
hpmcounter5	User-mode Hardware Performance Counter 2	0xc05	U
hpmcounter6	User-mode Hardware Performance Counter 3	0xc06	U
hpmcounter7	User-mode Hardware Performance Counter 4	0xc07	U
hpmcounter8	User-mode Hardware Performance Counter 5	0xc08	U
hpmcounter9	User-mode Hardware Performance Counter 6	0xc09	U

Appendix B: Instruction Details

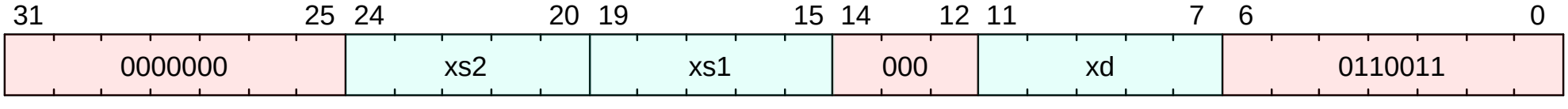
B.1. add

Integer add

This instruction is defined by:

I, version >= I@2.1.0

B.1.1. Encoding



B.1.2. Description

Add the value in xs1 to xs2, and store the result in xd. Any overflow is thrown away.

B.1.3. Access

M
Always

B.1.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.1.5. IDL Operation

```
X[xd] = X[xs1] + X[xs2];
```

B.1.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.1.7. Exceptions

This instruction does not generate synchronous exceptions.

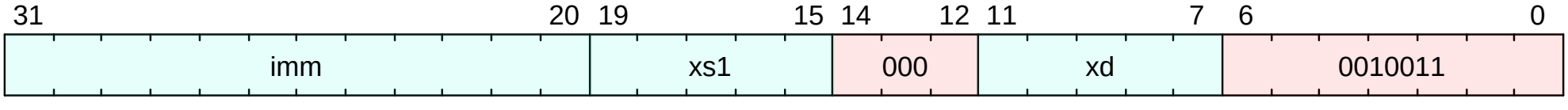
B.2. addi

Add immediate

This instruction is defined by:

I, version >= I@2.1.0

B.2.1. Encoding



B.2.2. Description

Adds an immediate value to the value in xs1, and store the result in xd

B.2.3. Access

M
Always

B.2.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.2.5. IDL Operation

```
X[xd] = X[xs1] + $signed(imm);
```

B.2.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => xs1_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(xs1_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(xs1_val <_u immext)),
    RISCV_ANDI  => xs1_val & immext,
    RISCV_ORI   => xs1_val | immext,
    RISCV_XORI  => xs1_val ^ immext
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.2.7. Exceptions

This instruction does not generate synchronous exceptions.

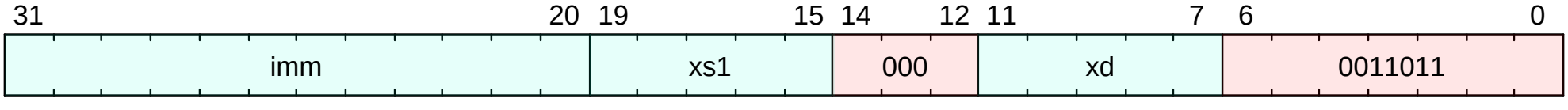
B.3. addiw

Add immediate word

This instruction is defined by:

I, version >= I@2.1.0

B.3.1. Encoding



B.3.2. Description

Add an immediate to the 32-bit value in xs1, and store the sign extended result in xd

B.3.3. Access

M
Always

B.3.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.3.5. IDL Operation

```
XReg operand = sext(X[xs1], 31);
X[xd] = sext(operand + imm, 31);
```

B.3.6. Sail Operation

```
{
  let result : xlenbits = sign_extend(imm) + X(xs1);
  X(xd) = sign_extend(result[31..0]);
  RETIRE_SUCCESS
}
```

B.3.7. Exceptions

This instruction does not generate synchronous exceptions.

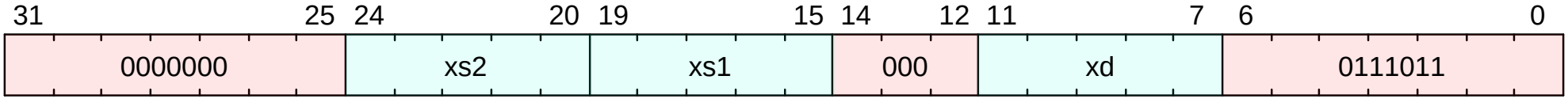
B.4. addw

Add word

This instruction is defined by:

I, version >= I@2.1.0

B.4.1. Encoding



B.4.2. Description

Add the 32-bit values in xs1 to xs2, and store the sign-extended result in xd. Any overflow is thrown away.

B.4.3. Access

M
Always

B.4.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.4.5. IDL Operation

```
XReg operand1 = sext(X[xs1], 31);
XReg operand2 = sext(X[xs2], 31);
X[xd] = sext(operand1 + operand2, 31);
```

B.4.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISCV_ADDW => xs1_val + xs2_val,
    RISCV_SUBW => xs1_val - xs2_val,
    RISCV_SLLW => xs1_val << (xs2_val[4..0]),
    RISCV_SRLW => xs1_val >> (xs2_val[4..0]),
    RISCV_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.4.7. Exceptions

This instruction does not generate synchronous exceptions.

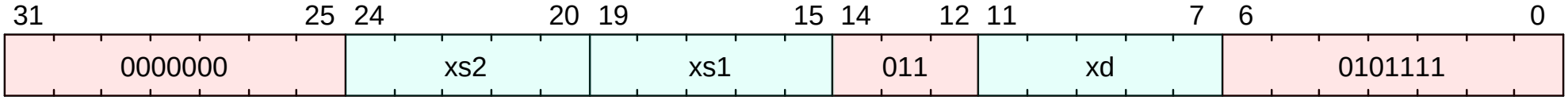
B.5. amoadd.d

Atomic fetch-and-add doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.5.1. Encoding



B.5.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Add the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.5.3. Access

M
Always

B.5.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.5.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Add, 1'b0, 1'b0, $encoding);
```

B.5.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.5.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

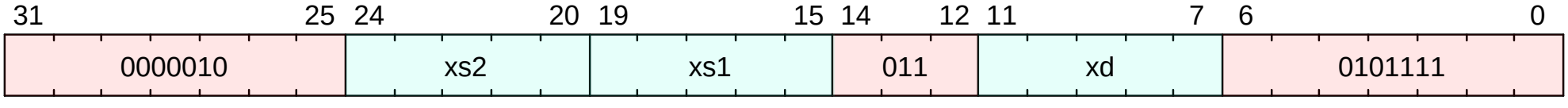
B.6. amoadd.d.aq

Atomic fetch-and-add doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.6.1. Encoding



B.6.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Add the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.6.3. Access

M
Always

B.6.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.6.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Add, 1'b1, 1'b0, $encoding);
```

B.6.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.6.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

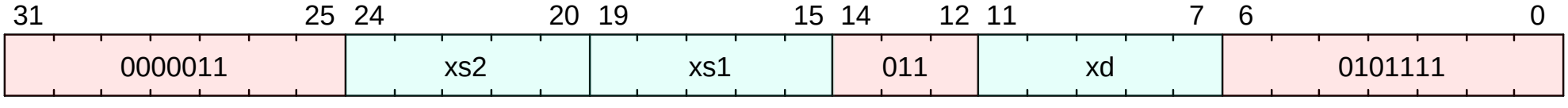
B.7. amoadd.d.aqrl

Atomic fetch-and-add doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.7.1. Encoding



B.7.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Add the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.7.3. Access

M
Always

B.7.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.7.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Add, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.7.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.7.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

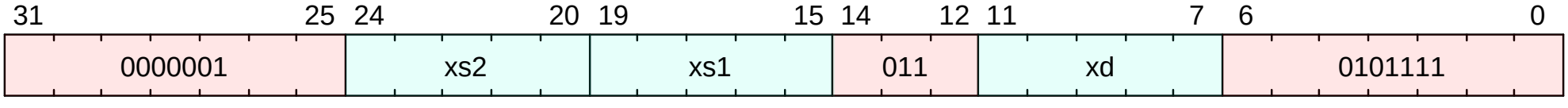
B.8. amoadd.d.rl

Atomic fetch-and-add doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.8.1. Encoding



B.8.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Add the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.8.3. Access

M
Always

B.8.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.8.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Add, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.8.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.8.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

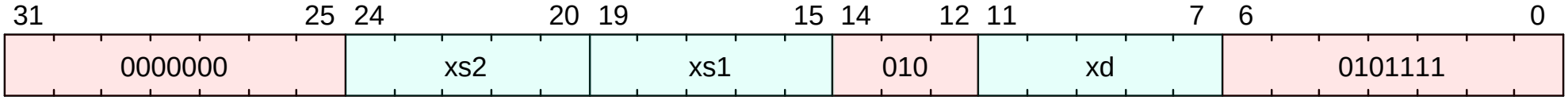
B.9. amoadd.w

Atomic fetch-and-add word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.9.1. Encoding



B.9.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Add the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.9.3. Access

M
Always

B.9.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.9.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Add, 1'b0, 1'b0, $encoding);
```

B.9.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            } else {
              handle_illegal();
              RETIRE_FAIL
            }
          }
```

B.9.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

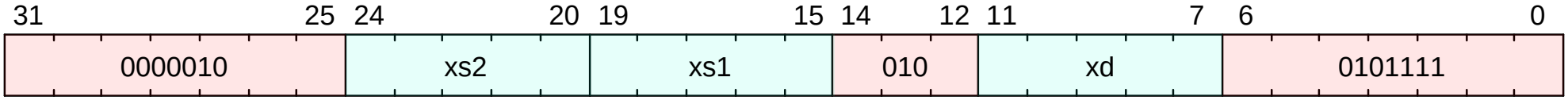
B.10. amoadd.w.aq

Atomic fetch-and-add word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.10.1. Encoding



B.10.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Add the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.10.3. Access

M
Always

B.10.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.10.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Add, 1'b1, 1'b0, $encoding);
```

B.10.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.10.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

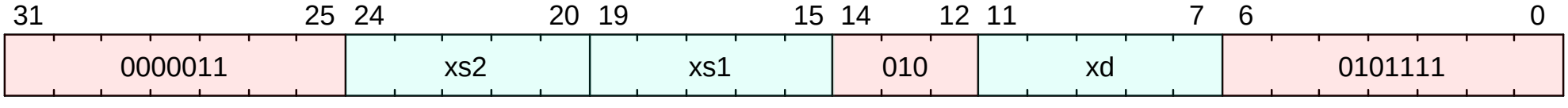
B.11. amoadd.w.aqrl

Atomic fetch-and-add word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.11.1. Encoding



B.11.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Add the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.11.3. Access

M
Always

B.11.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.11.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Add, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.11.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.11.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

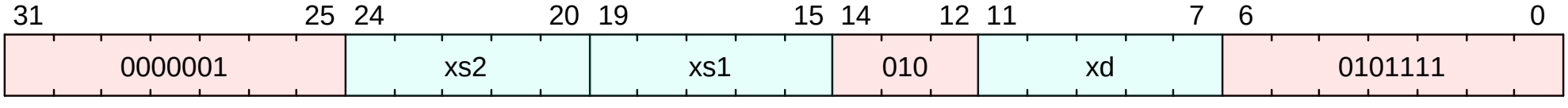
B.12. amoadd.w.rl

Atomic fetch-and-add word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.12.1. Encoding



B.12.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Add the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.12.3. Access

M
Always

B.12.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.12.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Add, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.12.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.12.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

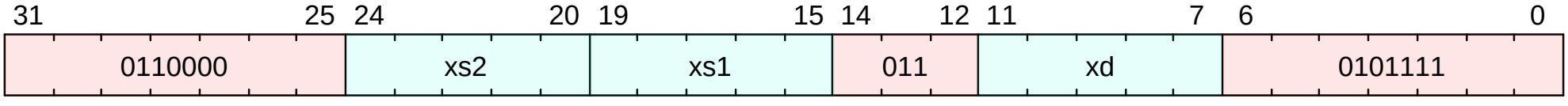
B.13. amoand.d

Atomic fetch-and-and doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.13.1. Encoding



B.13.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- AND the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.13.3. Access

M
Always

B.13.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.13.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::And, 1'b0, 1'b0, $encoding);
```

B.13.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.13.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

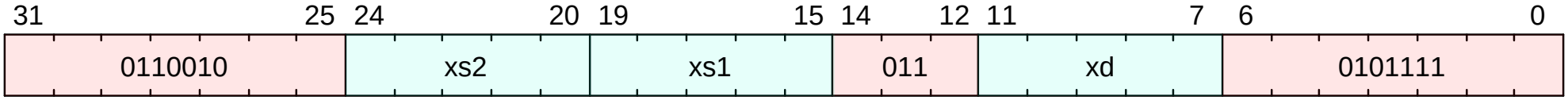
B.14. amoand.d.aq

Atomic fetch-and-and doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.14.1. Encoding



B.14.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- AND the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.14.3. Access

M
Always

B.14.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.14.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::And, 1'b1, 1'b0, $encoding);
```

B.14.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.14.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

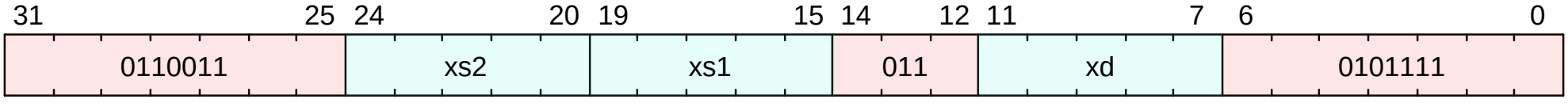
B.15. amoand.d.aqrl

Atomic fetch-and-and doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.15.1. Encoding



B.15.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- AND the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.15.3. Access

M
Always

B.15.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.15.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::And, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.15.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.15.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

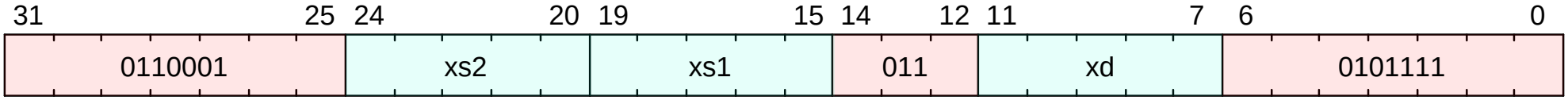
B.16. amoand.d.rl

Atomic fetch-and-and doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.16.1. Encoding



B.16.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- AND the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.16.3. Access

M
Always

B.16.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.16.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::And, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.16.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.16.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

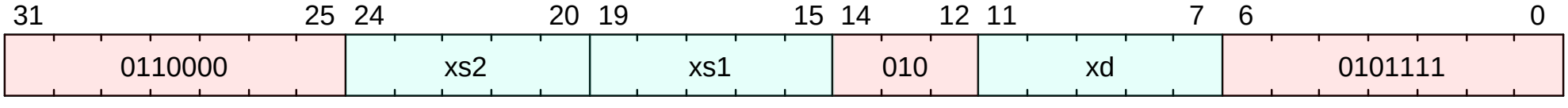
B.17. amoand.w

Atomic fetch-and-and word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.17.1. Encoding



B.17.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- AND the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.17.3. Access

M
Always

B.17.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.17.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::And, 1'b0, 1'b0, $encoding);
```

B.17.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.17.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

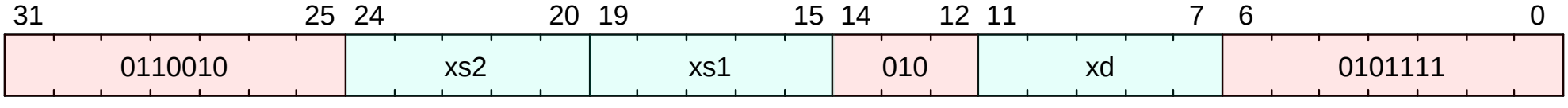
B.18. amoand.w.aq

Atomic fetch-and-and word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.18.1. Encoding



B.18.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- AND the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.18.3. Access

M
Always

B.18.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.18.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::And, 1'b1, 1'b0, $encoding);
```

B.18.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.18.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

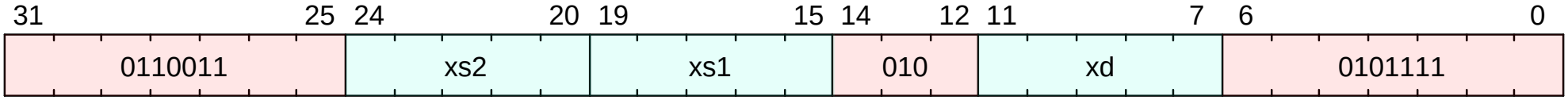
B.19. amoand.w.aqrl

Atomic fetch-and-and word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.19.1. Encoding



B.19.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- AND the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.19.3. Access

M
Always

B.19.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.19.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::And, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.19.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
      _       => false
    };
    let rs2_val : xlenbits = match width {
      BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
      HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
      WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
      DOUBLE => X(rs2)
    };
    match (eares) {
      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      MemValue(_) => {
        let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
          (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
          (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
          (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
          (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
          _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
        };
        match (mval) {
          MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(loaded) => {
            let result : xlenbits =
              match op {
                AMOSWAP => rs2_val,
                AMOADD  => rs2_val + loaded,
                AMOXOR  => rs2_val ^ loaded,
                AMOAND  => rs2_val & loaded,
                AMOOR   => rs2_val | loaded,

                /* These operations convert bitvectors to integer values using [un]signed,
                 * and back using to_bits().
                 */
                AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
              };
            let rval : xlenbits = match width {
              BYTE   => sign_extend(loaded[7..0]),
              HALF   => sign_extend(loaded[15..0]),
              WORD   => sign_extend(loaded[31..0]),
              DOUBLE => loaded
            };
            let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
              (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
              (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (wval) {
              MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
              MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
            }
          }
        }
      }
    } else {
      handle_illegal();
      RETIRE_FAIL
    }
  }
```

B.19.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

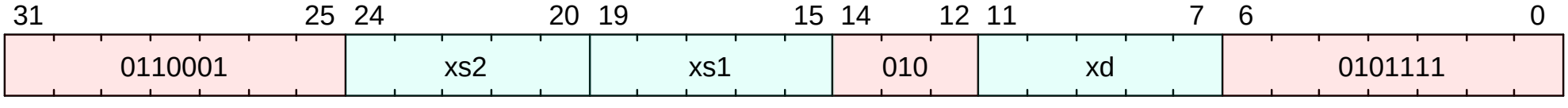
B.20. amoand.w.rl

Atomic fetch-and-and word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.20.1. Encoding



B.20.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- AND the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.20.3. Access

M
Always

B.20.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.20.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::And, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.20.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.20.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

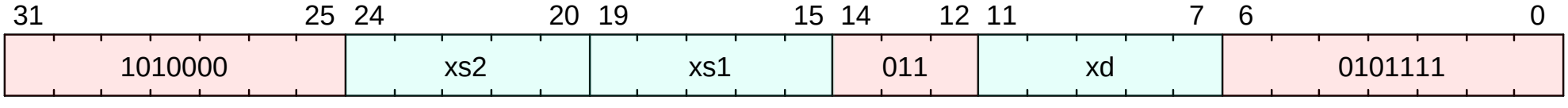
B.21. amomax.d

Atomic MAX doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.21.1. Encoding



B.21.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.21.3. Access

M
Always

B.21.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.21.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Max, 1'b0, 1'b0, $encoding);
```

B.21.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.21.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

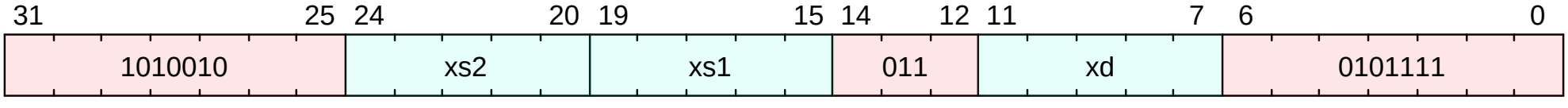
B.22. amomax.d.aq

Atomic MAX doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.22.1. Encoding



B.22.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.22.3. Access

M
Always

B.22.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.22.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Max, 1'b1, 1'b0, $encoding);
```

B.22.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.22.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

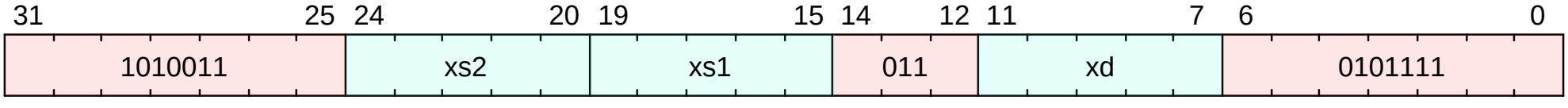
B.23. amomax.d.aqrl

Atomic MAX doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.23.1. Encoding



B.23.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.23.3. Access

M
Always

B.23.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.23.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Max, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.23.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.23.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

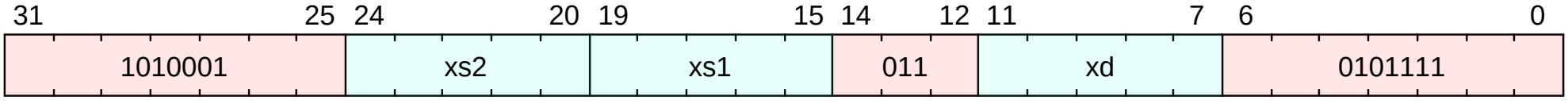
B.24. amomax.d.rl

Atomic MAX doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.24.1. Encoding



B.24.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.24.3. Access

M
Always

B.24.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.24.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Max, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.24.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.24.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

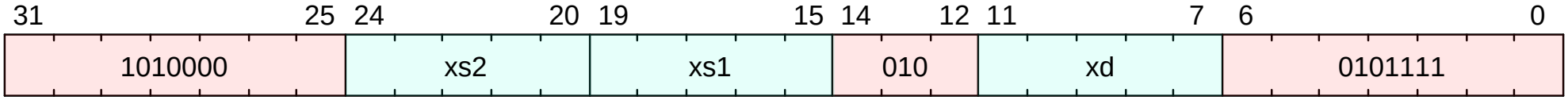
B.25. amomax.w

Atomic MAX word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.25.1. Encoding



B.25.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.25.3. Access

M
Always

B.25.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.25.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Max, 1'b0, 1'b0, $encoding);
```

B.25.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            } else {
              handle_illegal();
              RETIRE_FAIL
            }
          }
```

B.25.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

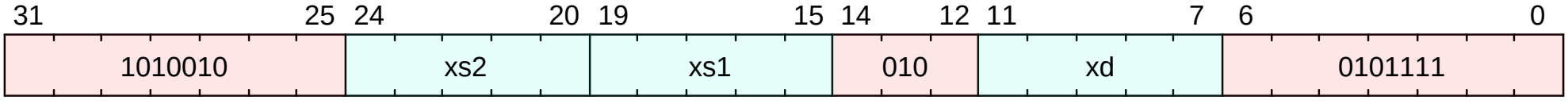
B.26. amomax.w.aq

Atomic MAX word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.26.1. Encoding



B.26.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.26.3. Access

M
Always

B.26.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.26.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Max, 1'b1, 1'b0, $encoding);
```

B.26.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.26.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

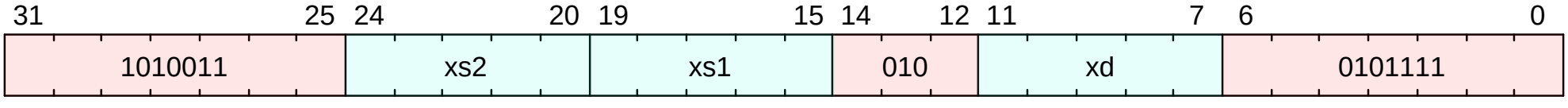
B.27. amomax.w.aqrl

Atomic MAX word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.27.1. Encoding



B.27.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.27.3. Access

M
Always

B.27.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.27.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Max, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.27.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.27.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

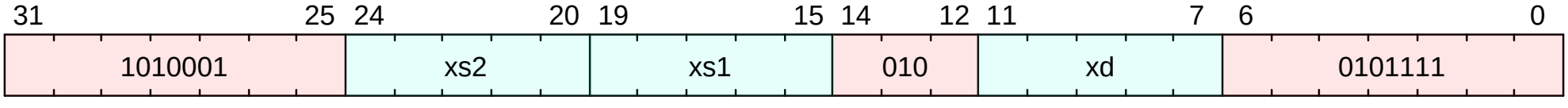
B.28. amomax.w.rl

Atomic MAX word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.28.1. Encoding



B.28.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.28.3. Access

M
Always

B.28.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.28.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Max, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.28.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.28.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

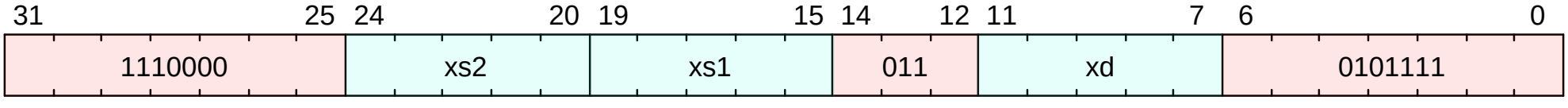
B.29. amomaxu.d

Atomic MAX unsigned doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.29.1. Encoding



B.29.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.29.3. Access

M
Always

B.29.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.29.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Maxu, 1'b0, 1'b0, $encoding);
```

B.29.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, false & false, false, true),
              (HALF, _)    => mem_write_ea(addr, 2, false & false, false, true),
              (WORD, _)    => mem_write_ea(addr, 4, false & false, false, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, false & false, false, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let xs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, false, false & false, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, false, false & false, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, false, false & false, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, false, false & false, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => xs2_val,
                        AMOADD  => xs2_val + loaded,
                        AMOXOR  => xs2_val ^ loaded,
                        AMOAND  => xs2_val & loaded,
                        AMOOR   => xs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  false & false, false, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], false & false, false, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], false & false, false, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        false & false, false, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.29.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

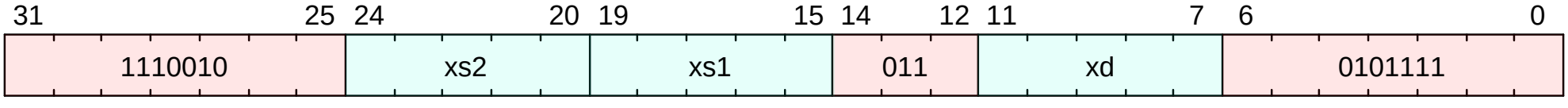
B.30. amomaxu.d.aq

Atomic MAX unsigned doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.30.1. Encoding



B.30.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.30.3. Access

M
Always

B.30.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.30.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Maxu, 1'b1, 1'b0, $encoding);
```

B.30.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, true & false, false, true),
              (HALF, _)    => mem_write_ea(addr, 2, true & false, false, true),
              (WORD, _)    => mem_write_ea(addr, 4, true & false, false, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, true & false, false, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let xs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, true, true & false, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, true, true & false, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, true, true & false, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, true, true & false, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => xs2_val,
                    AMOADD  => xs2_val + loaded,
                    AMOXOR  => xs2_val ^ loaded,
                    AMOAND  => xs2_val & loaded,
                    AMOOR   => xs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  true & false, false, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], true & false, false, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], true & false, false, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        true & false, false, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.30.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

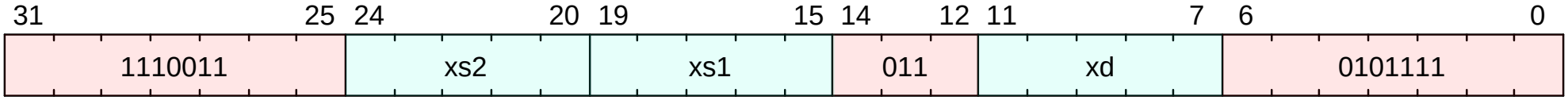
B.31. amomaxu.d.aqrl

Atomic MAX unsigned doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.31.1. Encoding



B.31.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.31.3. Access

M
Always

B.31.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.31.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Maxu, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.31.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, true & true, true, true),
              (HALF, _)    => mem_write_ea(addr, 2, true & true, true, true),
              (WORD, _)    => mem_write_ea(addr, 4, true & true, true, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, true & true, true, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let xs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, true, true & true, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, true, true & true, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, true, true & true, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, true, true & true, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => xs2_val,
                  AMOADD  => xs2_val + loaded,
                  AMOXOR  => xs2_val ^ loaded,
                  AMOAND  => xs2_val & loaded,
                  AMOOR   => xs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  true & true, true, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], true & true, true, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], true & true, true, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        true & true, true, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.31.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

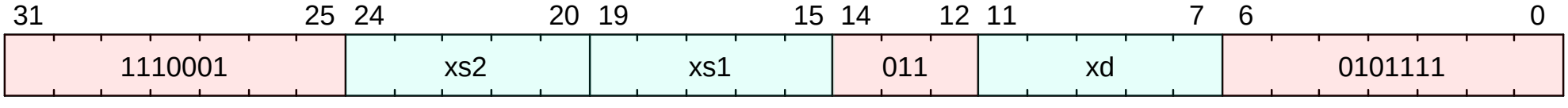
B.32. amomaxu.d.rl

Atomic MAX unsigned doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.32.1. Encoding



B.32.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.32.3. Access

M
Always

B.32.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.32.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Maxu, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.32.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, false & true, true, true),
              (HALF, _)    => mem_write_ea(addr, 2, false & true, true, true),
              (WORD, _)    => mem_write_ea(addr, 4, false & true, true, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, false & true, true, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let xs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, false, false & true, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, false, false & true, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, false, false & true, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, false, false & true, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => xs2_val,
                        AMOADD  => xs2_val + loaded,
                        AMOXOR  => xs2_val ^ loaded,
                        AMOAND  => xs2_val & loaded,
                        AMOOR   => xs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  false & true, true, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], false & true, true, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], false & true, true, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        false & true, true, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.32.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

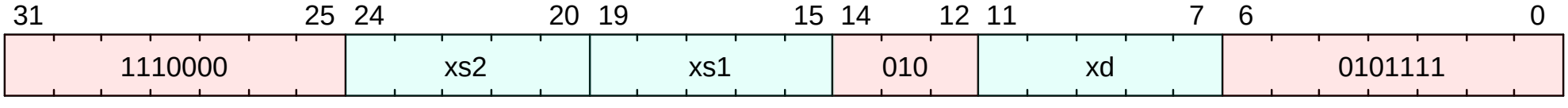
B.33. amomaxu.w

Atomic MAX unsigned word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.33.1. Encoding



B.33.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.33.3. Access

M
Always

B.33.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.33.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Maxu, 1'b0, 1'b0, $encoding);
```

B.33.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, false & false, false, true),
              (HALF, _)    => mem_write_ea(addr, 2, false & false, false, true),
              (WORD, _)    => mem_write_ea(addr, 4, false & false, false, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, false & false, false, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let xs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, false, false & false, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, false, false & false, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, false, false & false, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, false, false & false, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => xs2_val,
                        AMOADD  => xs2_val + loaded,
                        AMOXOR  => xs2_val ^ loaded,
                        AMOAND  => xs2_val & loaded,
                        AMOOR   => xs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  false & false, false, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], false & false, false, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], false & false, false, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        false & false, false, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.33.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

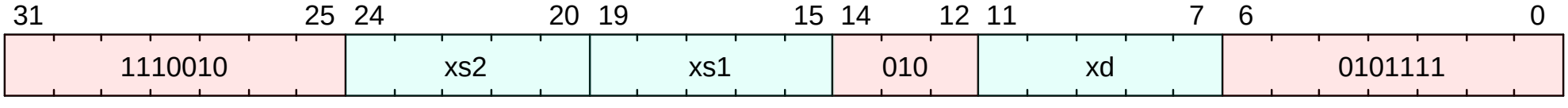
B.34. amomaxu.w.aq

Atomic MAX unsigned word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.34.1. Encoding



B.34.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.34.3. Access

M
Always

B.34.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.34.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Maxu, 1'b1, 1'b0, $encoding);
```

B.34.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, true & false, false, true),
              (HALF, _)    => mem_write_ea(addr, 2, true & false, false, true),
              (WORD, _)    => mem_write_ea(addr, 4, true & false, false, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, true & false, false, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let xs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, true, true & false, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, true, true & false, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, true, true & false, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, true, true & false, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => xs2_val,
                        AMOADD  => xs2_val + loaded,
                        AMOXOR  => xs2_val ^ loaded,
                        AMOAND  => xs2_val & loaded,
                        AMOOR   => xs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  true & false, false, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], true & false, false, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], true & false, false, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        true & false, false, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.34.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

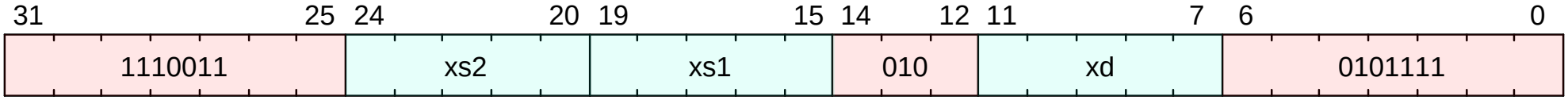
B.35. amomaxu.w.aqrl

Atomic MAX unsigned word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.35.1. Encoding



B.35.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.35.3. Access

M
Always

B.35.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.35.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Maxu, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.35.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, true & true, true, true),
              (HALF, _)    => mem_write_ea(addr, 2, true & true, true, true),
              (WORD, _)    => mem_write_ea(addr, 4, true & true, true, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, true & true, true, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let xs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, true, true & true, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, true, true & true, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, true, true & true, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, true, true & true, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => xs2_val,
                  AMOADD  => xs2_val + loaded,
                  AMOXOR  => xs2_val ^ loaded,
                  AMOAND  => xs2_val & loaded,
                  AMOOR   => xs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  true & true, true, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], true & true, true, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], true & true, true, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        true & true, true, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.35.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

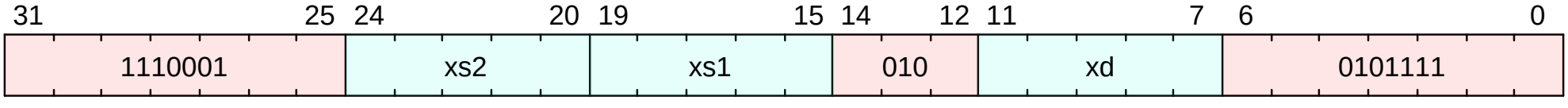
B.36. amomaxu.w.rl

Atomic MAX unsigned word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.36.1. Encoding



B.36.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *xs1*

B.36.3. Access

M
Always

B.36.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.36.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Maxu, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.36.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, false & true, true, true),
              (HALF, _)    => mem_write_ea(addr, 2, false & true, true, true),
              (WORD, _)    => mem_write_ea(addr, 4, false & true, true, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, false & true, true, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let xs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, false, false & true, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, false, false & true, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, false, false & true, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, false, false & true, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => xs2_val,
                        AMOADD  => xs2_val + loaded,
                        AMOXOR  => xs2_val ^ loaded,
                        AMOAND  => xs2_val & loaded,
                        AMOOR   => xs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(xs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(xs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(xs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(xs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  false & true, true, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], false & true, true, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], false & true, true, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        false & true, true, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.36.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

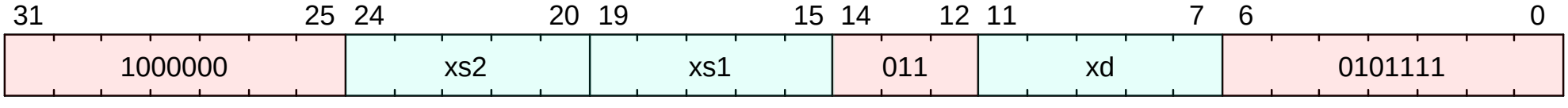
B.37. amomin.d

Atomic MIN doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.37.1. Encoding



B.37.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.37.3. Access

M
Always

B.37.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.37.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Min, 1'b0, 1'b0, $encoding);
```

B.37.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.37.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

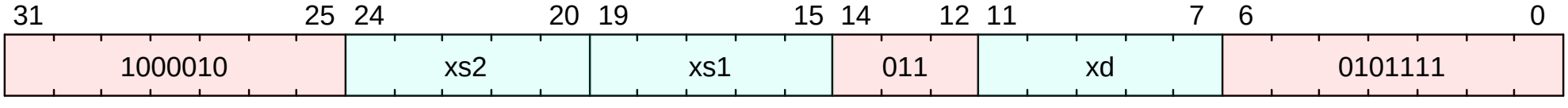
B.38. amomin.d.aq

Atomic MIN doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.38.1. Encoding



B.38.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.38.3. Access

M
Always

B.38.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.38.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Min, 1'b1, 1'b0, $encoding);
```

B.38.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.38.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

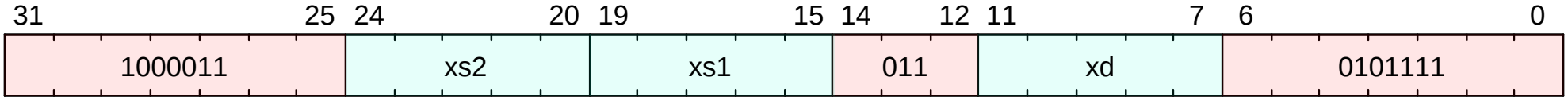
B.39. amomin.d.aqrl

Atomic MIN doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.39.1. Encoding



B.39.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.39.3. Access

M
Always

B.39.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.39.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Min, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.39.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.39.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

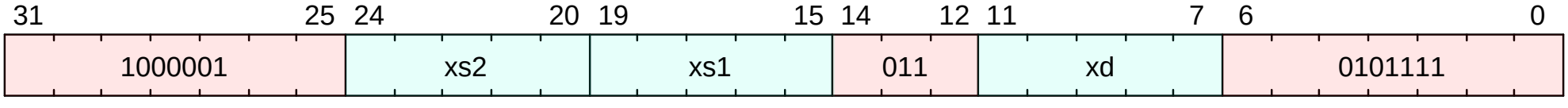
B.40. amomin.d.rl

Atomic MIN doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.40.1. Encoding



B.40.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Signed compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.40.3. Access

M
Always

B.40.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.40.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Min, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.40.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.40.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

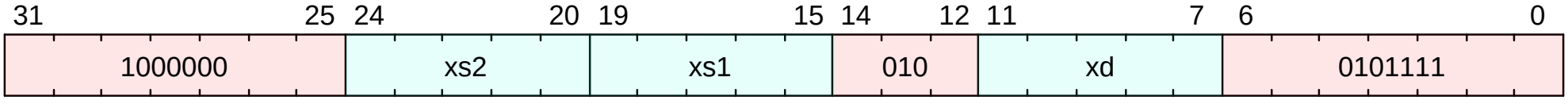
B.41. amomin.w

Atomic MIN word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.41.1. Encoding



B.41.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.41.3. Access

M
Always

B.41.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.41.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Min, 1'b0, 1'b0, $encoding);
```

B.41.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            } else {
              handle_illegal();
              RETIRE_FAIL
            }
          }
```

B.41.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

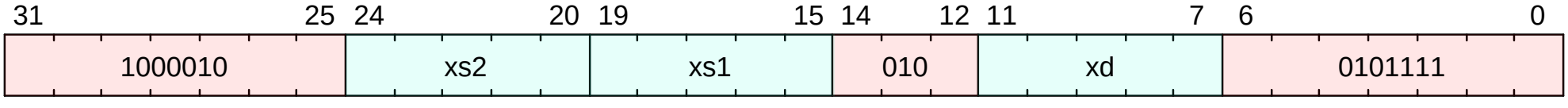
B.42. amomin.w.aq

Atomic MIN word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.42.1. Encoding



B.42.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.42.3. Access

M
Always

B.42.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.42.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Min, 1'b1, 1'b0, $encoding);
```

B.42.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.42.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

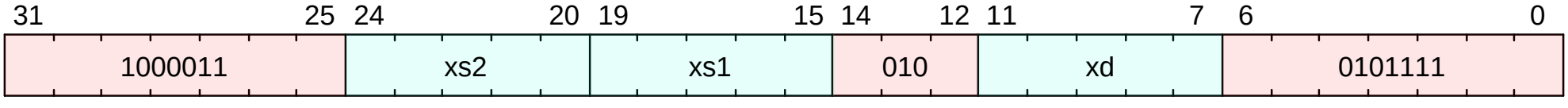
B.43. amomin.w.aqrl

Atomic MIN word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.43.1. Encoding



B.43.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.43.3. Access

M
Always

B.43.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.43.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Min, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.43.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.43.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

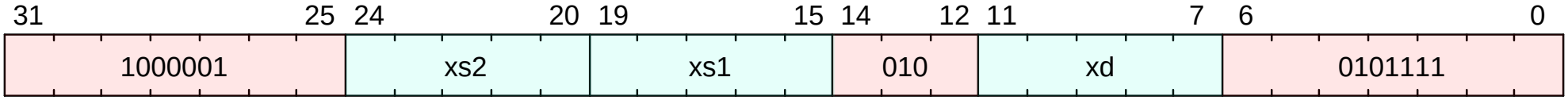
B.44. amomin.w.rl

Atomic MIN word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.44.1. Encoding



B.44.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Signed compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.44.3. Access

M
Always

B.44.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.44.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Min, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.44.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.44.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

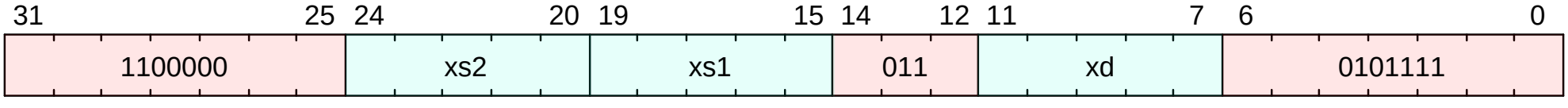
B.45. amominu.d

Atomic MIN unsigned doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.45.1. Encoding



B.45.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.45.3. Access

M
Always

B.45.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.45.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Minu, 1'b0, 1'b0, $encoding);
```

B.45.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.45.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

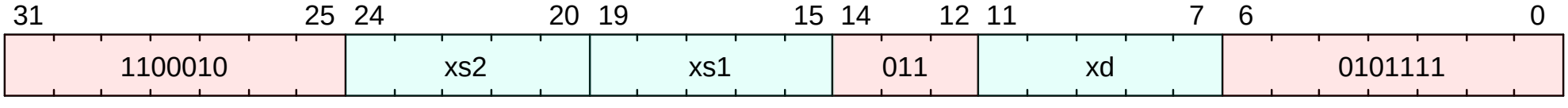
B.46. amominu.d.aq

Atomic MIN unsigned doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.46.1. Encoding



B.46.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.46.3. Access

M
Always

B.46.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.46.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Minu, 1'b1, 1'b0, $encoding);
```

B.46.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.46.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

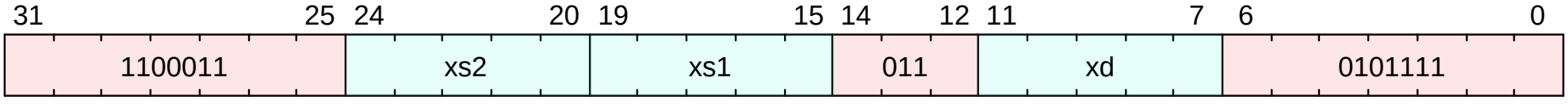
B.47. amominu.d.aqrl

Atomic MIN unsigned doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.47.1. Encoding



B.47.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.47.3. Access

M
Always

B.47.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.47.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Minu, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.47.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.47.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

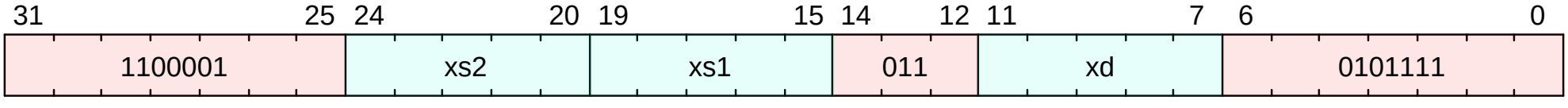
B.48. amominu.d.rl

Atomic MIN unsigned doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.48.1. Encoding



B.48.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Unsigned compare the value of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.48.3. Access

M
Always

B.48.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.48.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Minu, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.48.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.48.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

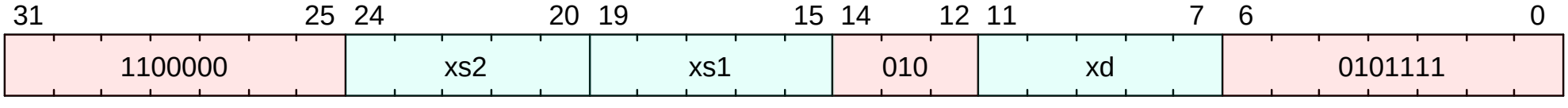
B.49. amominu.w

Atomic MIN unsigned word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.49.1. Encoding



B.49.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.49.3. Access

M
Always

B.49.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.49.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Minu, 1'b0, 1'b0, $encoding);
```

B.49.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.49.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

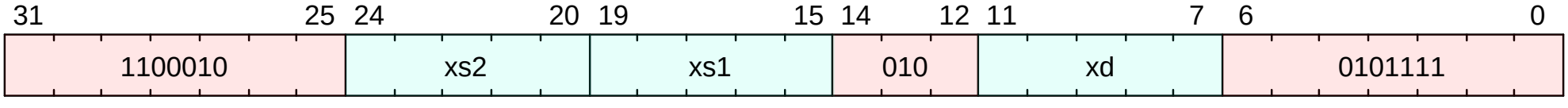
B.50. amominu.w.aq

Atomic MIN unsigned word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.50.1. Encoding



B.50.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.50.3. Access

M
Always

B.50.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.50.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Minu, 1'b1, 1'b0, $encoding);
```

B.50.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.50.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

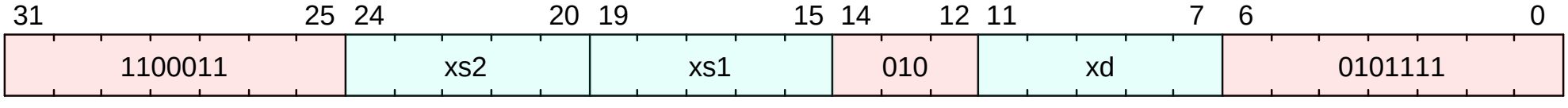
B.51. amominu.w.aqrl

Atomic MIN unsigned word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.51.1. Encoding



B.51.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.51.3. Access

M
Always

B.51.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.51.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Minu, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.51.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.51.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

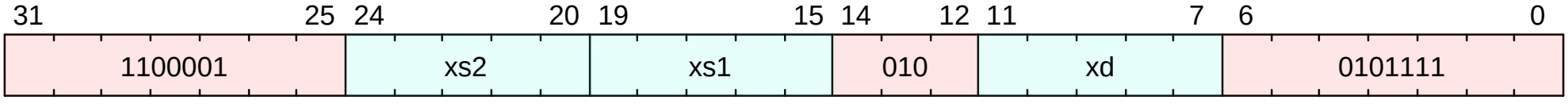
B.52. amominu.w.rl

Atomic MIN unsigned word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.52.1. Encoding



B.52.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Unsigned compare the least-significant word of register *xs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *xs1*

B.52.3. Access

M
Always

B.52.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.52.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Minu, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.52.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.52.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

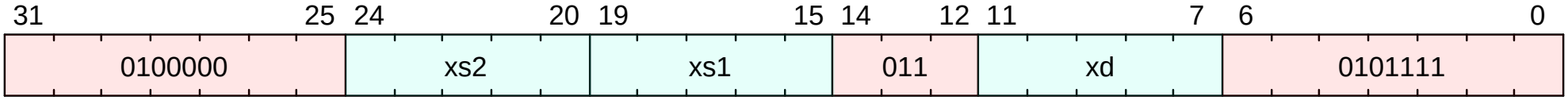
B.53. amoor.d

Atomic fetch-and-or doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.53.1. Encoding



B.53.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- OR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.53.3. Access

M
Always

B.53.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.53.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Or, 1'b0, 1'b0, $encoding);
```

B.53.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            } else {
              handle_illegal();
              RETIRE_FAIL
            }
          }
```

B.53.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

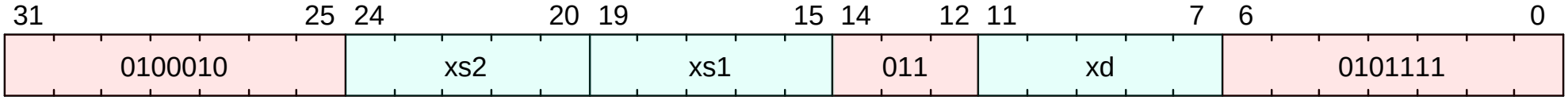
B.54. amoor.d.aq

Atomic fetch-and-or doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.54.1. Encoding



B.54.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- OR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.54.3. Access

M
Always

B.54.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.54.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Or, 1'b1, 1'b0, $encoding);
```

B.54.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.54.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

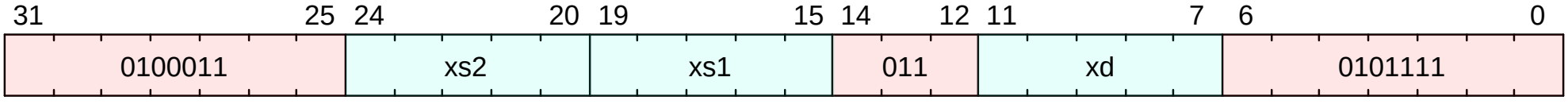
B.55. amoor.d.aqrl

Atomic fetch-and-or doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.55.1. Encoding



B.55.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- OR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.55.3. Access

M
Always

B.55.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.55.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Or, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.55.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
      _       => false
    };
    let rs2_val : xlenbits = match width {
      BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
      HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
      WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
      DOUBLE => X(rs2)
    };
    match (eares) {
      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      MemValue(_) => {
        let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
          (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
          (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
          (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
          (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
          _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
        };
        match (mval) {
          MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(loaded) => {
            let result : xlenbits =
              match op {
                AMOSWAP => rs2_val,
                AMOADD  => rs2_val + loaded,
                AMOXOR  => rs2_val ^ loaded,
                AMOAND  => rs2_val & loaded,
                AMOOR   => rs2_val | loaded,

                /* These operations convert bitvectors to integer values using [un]signed,
                 * and back using to_bits().
                 */
                AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
              };
            let rval : xlenbits = match width {
              BYTE   => sign_extend(loaded[7..0]),
              HALF   => sign_extend(loaded[15..0]),
              WORD   => sign_extend(loaded[31..0]),
              DOUBLE => loaded
            };
            let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
              (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
              (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (wval) {
              MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
              MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
            }
          }
        }
      }
    } else {
      handle_illegal();
      RETIRE_FAIL
    }
  }
```

B.55.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

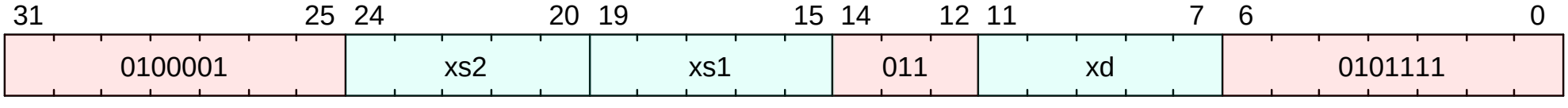
B.56. amoor.d.rl

Atomic fetch-and-or doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.56.1. Encoding



B.56.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- OR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.56.3. Access

M
Always

B.56.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.56.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Or, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.56.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.56.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

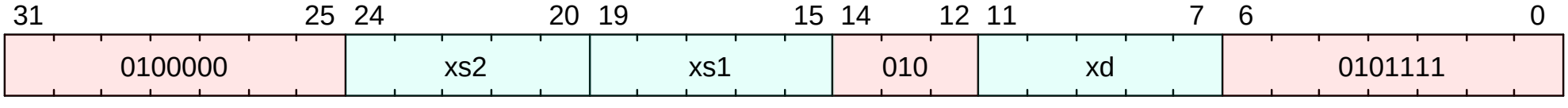
B.57. amoor.w

Atomic fetch-and-or word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.57.1. Encoding



B.57.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- OR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.57.3. Access

M
Always

B.57.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.57.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Or, 1'b0, 1'b0, $encoding);
```

B.57.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.57.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

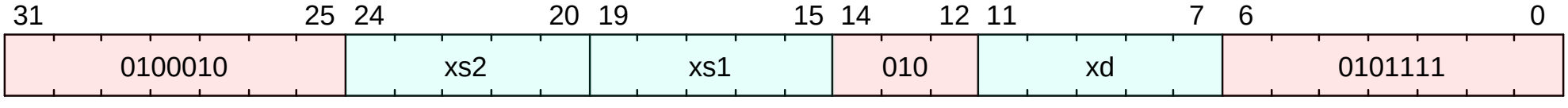
B.58. amoor.w.aq

Atomic fetch-and-or word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.58.1. Encoding



B.58.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- OR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.58.3. Access

M
Always

B.58.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.58.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Or, 1'b1, 1'b0, $encoding);
```

B.58.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.58.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

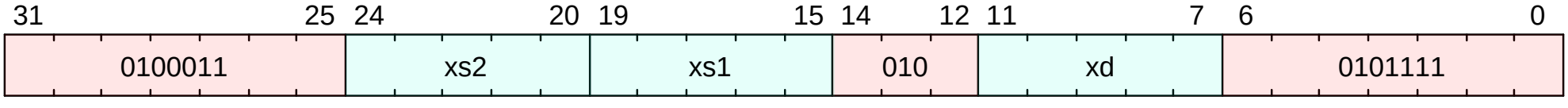
B.59. amoor.w.aqrl

Atomic fetch-and-or word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.59.1. Encoding



B.59.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- OR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.59.3. Access

M
Always

B.59.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.59.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Or, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.59.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
            _       => false
          };
          let rs2_val : xlenbits = match width {
            BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
            HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
            WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
            DOUBLE => X(rs2)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (mval) {
                MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                MemValue(loaded) => {
                  let result : xlenbits =
                    match op {
                      AMOSWAP => rs2_val,
                      AMOADD  => rs2_val + loaded,
                      AMOXOR  => rs2_val ^ loaded,
                      AMOAND  => rs2_val & loaded,
                      AMOOR   => rs2_val | loaded,

                      /* These operations convert bitvectors to integer values using [un]signed,
                       * and back using to_bits().
                       */
                      AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                      AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                      AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                      AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                    };
                  let rval : xlenbits = match width {
                    BYTE   => sign_extend(loaded[7..0]),
                    HALF   => sign_extend(loaded[15..0]),
                    WORD   => sign_extend(loaded[31..0]),
                    DOUBLE => loaded
                  };
                  let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                    (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                    (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                  };
                  match (wval) {
                    MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                    MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.59.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

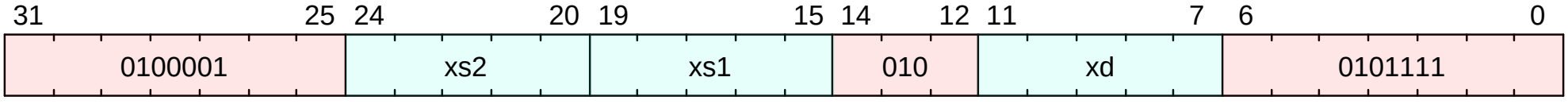
B.60. amoor.w.rl

Atomic fetch-and-or word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.60.1. Encoding



B.60.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- OR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.60.3. Access

M
Always

B.60.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.60.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Or, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.60.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.60.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

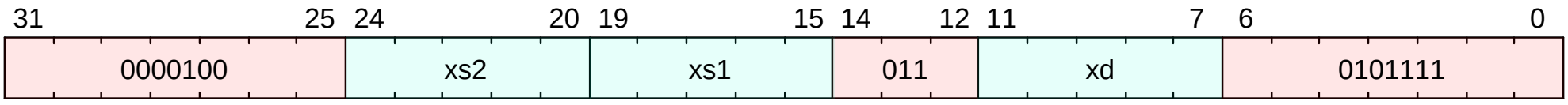
B.61. amoswap.d

Atomic SWAP doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.61.1. Encoding



B.61.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Store the value of register *xs2* to the address in *xs1*

B.61.3. Access

M
Always

B.61.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.61.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Swap, 1'b0, 1'b0, $encoding);
```

B.61.6. Sail Operation

```
{
  if extension("A") then {
  /* Get the address, X(rs1) (no offset).
   * Some extensions perform additional checks on address validity.
   */
  match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) => {
      match translateAddr(vaddr, ReadWrite(Data, Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(addr, _) => {
          let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
            (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
            (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
            (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
            (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          let is_unsigned : bool = match op {
            AMOMINU => true,
            AMOMAXU => true,
            _       => false
          };
          let rs2_val : xlenbits = match width {
            BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
```

```
            HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
            WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
            DOUBLE => X(rs2)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (mval) {
                MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                MemValue(loaded) => {
                  let result : xlenbits =
                    match op {
                      AMOSWAP => rs2_val,
                      AMOADD  => rs2_val + loaded,
                      AMOXOR  => rs2_val ^ loaded,
                      AMOAND  => rs2_val & loaded,
                      AMOOR   => rs2_val | loaded,
                      /* These operations convert bitvectors to integer values using [un]signed,
                       * and back using to_bits().
                       */
                      AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                      AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                      AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                      AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                    };
                  let rval : xlenbits = match width {
                    BYTE   => sign_extend(loaded[7..0]),
                    HALF   => sign_extend(loaded[15..0]),
                    WORD   => sign_extend(loaded[31..0]),
                    DOUBLE => loaded
                  };
                  let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                    (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                    (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                  };
                  match (wval) {
                    MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                    MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.61.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

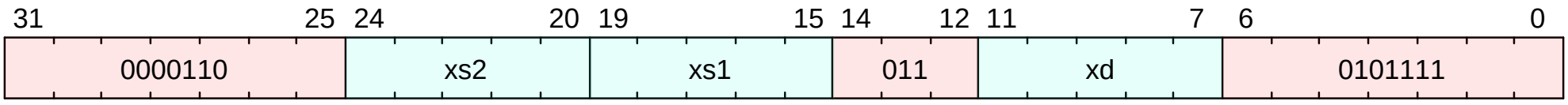
B.62. amoswap.d.aq

Atomic SWAP doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.62.1. Encoding



B.62.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Store the value of register *xs2* to the address in *xs1*

B.62.3. Access

M
Always

B.62.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.62.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Swap, 1'b1, 1'b0, $encoding);
```

B.62.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
            let rs2_val : xlenbits = match width {
```

```
            BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
            HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
            WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
            DOUBLE => X(rs2)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (mval) {
                MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                MemValue(loaded) => {
                  let result : xlenbits =
                    match op {
                      AMOSWAP => rs2_val,
                      AMOADD  => rs2_val + loaded,
                      AMOXOR  => rs2_val ^ loaded,
                      AMOAND  => rs2_val & loaded,
                      AMOOR   => rs2_val | loaded,

                      /* These operations convert bitvectors to integer values using [un]signed,
                       * and back using to_bits().
                       */
                      AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                      AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                      AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                      AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                    };
                  let rval : xlenbits = match width {
                    BYTE   => sign_extend(loaded[7..0]),
                    HALF   => sign_extend(loaded[15..0]),
                    WORD   => sign_extend(loaded[31..0]),
                    DOUBLE => loaded
                  };
                  let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                    (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                    (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                  };
                  match (wval) {
                    MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                    MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.62.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

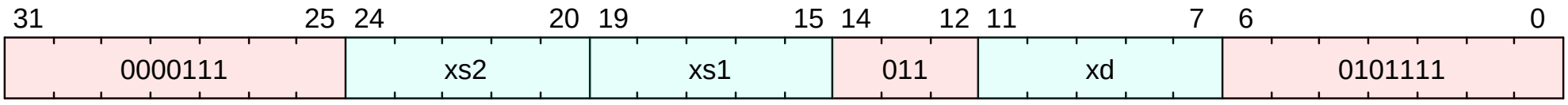
B.63. amoswap.d.aqrl

Atomic SWAP doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.63.1. Encoding



B.63.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Store the value of register *xs2* to the address in *xs1*

B.63.3. Access

M
Always

B.63.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.63.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Swap, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.63.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.63.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

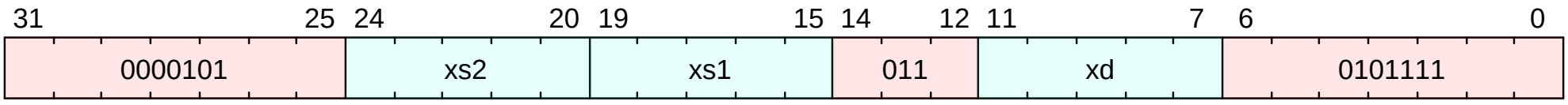
B.64. amoswap.d.rl

Atomic SWAP doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.64.1. Encoding



B.64.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- Store the value of register *xs2* to the address in *xs1*

B.64.3. Access

M
Always

B.64.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.64.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Swap, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.64.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
            let rs2_val : xlenbits = match width {
```

```
            BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
            HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
            WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
            DOUBLE => X(rs2)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (mval) {
                MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                MemValue(loaded) => {
                  let result : xlenbits =
                    match op {
                      AMOSWAP => rs2_val,
                      AMOADD  => rs2_val + loaded,
                      AMOXOR  => rs2_val ^ loaded,
                      AMOAND  => rs2_val & loaded,
                      AMOOR   => rs2_val | loaded,

                      /* These operations convert bitvectors to integer values using [un]signed,
                       * and back using to_bits().
                       */
                      AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                      AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                      AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                      AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                    };
                  let rval : xlenbits = match width {
                    BYTE   => sign_extend(loaded[7..0]),
                    HALF   => sign_extend(loaded[15..0]),
                    WORD   => sign_extend(loaded[31..0]),
                    DOUBLE => loaded
                  };
                  let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                    (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                    (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                  };
                  match (wval) {
                    MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                    MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                  }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.64.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

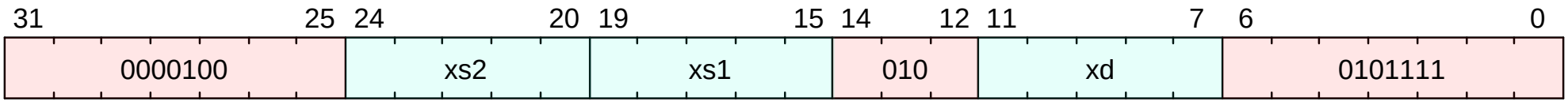
B.65. amoswap.w

Atomic SWAP word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.65.1. Encoding



B.65.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Store the least-significant word of register *xs2* to the address in *xs1*

B.65.3. Access

M
Always

B.65.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.65.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Swap, 1'b0, 1'b0, $encoding);
```

B.65.6. Sail Operation

```
{
  if extension("A") then {
  /* Get the address, X(rs1) (no offset).
   * Some extensions perform additional checks on address validity.
   */
  match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) => {
      match translateAddr(vaddr, ReadWrite(Data, Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(addr, _) => {
          let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
            (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
            (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
            (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
            (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          let is_unsigned : bool = match op {
            AMOMINU => true,
            AMOMAXU => true,
            _       => false
          };
          let rs2_val : xlenbits = match width {
            BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
```

```
            HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
            WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
            DOUBLE => X(rs2)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (mval) {
                MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                MemValue(loaded) => {
                  let result : xlenbits =
                    match op {
                      AMOSWAP => rs2_val,
                      AMOADD  => rs2_val + loaded,
                      AMOXOR  => rs2_val ^ loaded,
                      AMOAND  => rs2_val & loaded,
                      AMOOR   => rs2_val | loaded,
                      /* These operations convert bitvectors to integer values using [un]signed,
                       * and back using to_bits().
                       */
                      AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                      AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                      AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                      AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                    };
                  let rval : xlenbits = match width {
                    BYTE   => sign_extend(loaded[7..0]),
                    HALF   => sign_extend(loaded[15..0]),
                    WORD   => sign_extend(loaded[31..0]),
                    DOUBLE => loaded
                  };
                  let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                    (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                    (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                  };
                  match (wval) {
                    MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                    MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.65.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

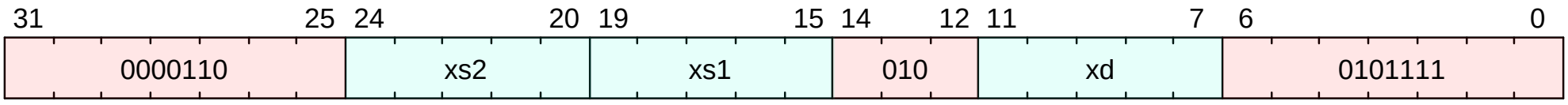
B.66. amoswap.w.aq

Atomic SWAP word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.66.1. Encoding



B.66.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Store the least-significant word of register *xs2* to the address in *xs1*

B.66.3. Access

M
Always

B.66.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.66.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Swap, 1'b1, 1'b0, $encoding);
```

B.66.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
            let rs2_val : xlenbits = match width {
```

```
            BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
            HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
            WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
            DOUBLE => X(rs2)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (mval) {
                MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                MemValue(loaded) => {
                  let result : xlenbits =
                    match op {
                      AMOSWAP => rs2_val,
                      AMOADD  => rs2_val + loaded,
                      AMOXOR  => rs2_val ^ loaded,
                      AMOAND  => rs2_val & loaded,
                      AMOOR   => rs2_val | loaded,

                      /* These operations convert bitvectors to integer values using [un]signed,
                       * and back using to_bits().
                       */
                      AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                      AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                      AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                      AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                    };
                  let rval : xlenbits = match width {
                    BYTE   => sign_extend(loaded[7..0]),
                    HALF   => sign_extend(loaded[15..0]),
                    WORD   => sign_extend(loaded[31..0]),
                    DOUBLE => loaded
                  };
                  let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                    (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                    (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                  };
                  match (wval) {
                    MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                    MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                  }
                }
              }
            }
          } else {
            handle_illegal();
            RETIRE_FAIL
          }
        }
```

B.66.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

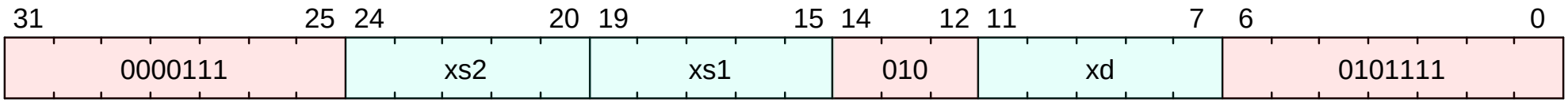
B.67. amoswap.w.aqrl

Atomic SWAP word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.67.1. Encoding



B.67.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Store the least-significant word of register *xs2* to the address in *xs1*

B.67.3. Access

M
Always

B.67.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.67.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Swap, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.67.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.67.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

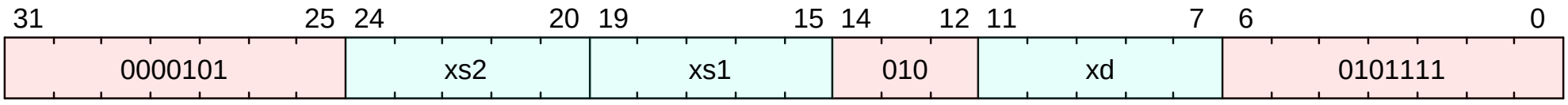
B.68. amoswap.w.rl

Atomic SWAP word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.68.1. Encoding



B.68.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- Store the least-significant word of register *xs2* to the address in *xs1*

B.68.3. Access

M
Always

B.68.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.68.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Swap, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.68.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
            let rs2_val : xlenbits = match width {
```

```
                  BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
                  HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
                  WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
                  DOUBLE => X(rs2)
                };
                match (eares) {
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(_) => {
                    let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                      (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                      (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                      (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (mval) {
                      MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                      MemValue(loaded) => {
                        let result : xlenbits =
                          match op {
                            AMOSWAP => rs2_val,
                            AMOADD  => rs2_val + loaded,
                            AMOXOR  => rs2_val ^ loaded,
                            AMOAND  => rs2_val & loaded,
                            AMOOR   => rs2_val | loaded,

                            /* These operations convert bitvectors to integer values using [un]signed,
                             * and back using to_bits().
                             */
                            AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                            AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                            AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                            AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                          };
                        let rval : xlenbits = match width {
                          BYTE   => sign_extend(loaded[7..0]),
                          HALF   => sign_extend(loaded[15..0]),
                          WORD   => sign_extend(loaded[31..0]),
                          DOUBLE => loaded
                        };
                        let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                          (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                          (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                          (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                          (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                          _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                        };
                        match (wval) {
                          MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                          MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                        }
                      }
                    }
                  }
                } else {
                  handle_illegal();
                  RETIRE_FAIL
                }
              }
```

B.68.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault

- StoreAmoAddressMisaligned
- StoreAmoPageFault

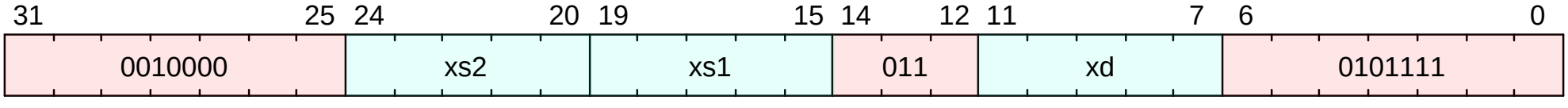
B.69. amoxor.d

Atomic fetch-and-xor doubleword

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.69.1. Encoding



B.69.2. Description

Atomically:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- XOR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.69.3. Access

M
Always

B.69.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.69.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Xor, 1'b0, 1'b0, $encoding);
```

B.69.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.69.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

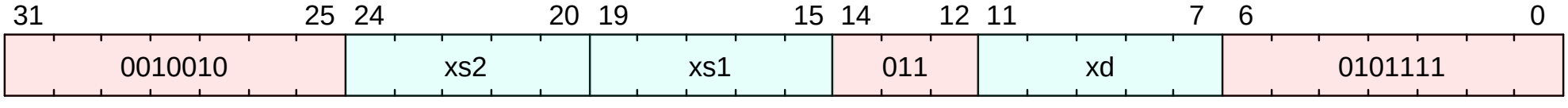
B.70. amoxor.d.aq

Atomic fetch-and-xor doubleword (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.70.1. Encoding



B.70.2. Description

Atomically with acquire ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- XOR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.70.3. Access

M
Always

B.70.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.70.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Xor, 1'b1, 1'b0, $encoding);
```

B.70.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.70.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

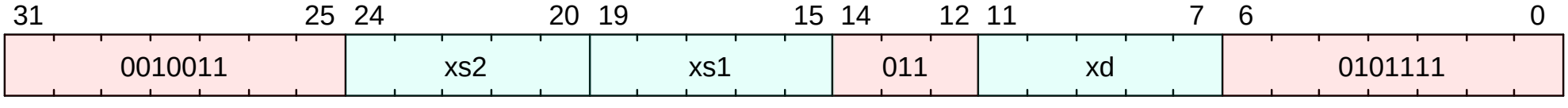
B.71. amoxor.d.aqrl

Atomic fetch-and-xor doubleword (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.71.1. Encoding



B.71.2. Description

Atomically with acquire and release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- XOR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.71.3. Access

M
Always

B.71.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.71.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Xor, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.71.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.71.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

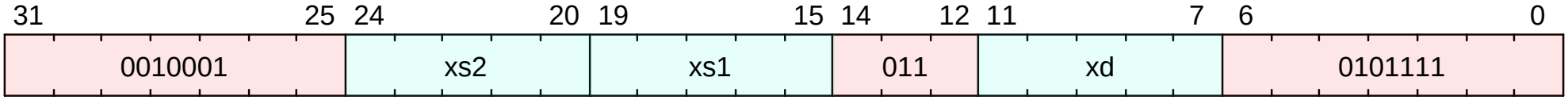
B.72. amoxor.d.rl

Atomic fetch-and-xor doubleword (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.72.1. Encoding



B.72.2. Description

Atomically with release ordering:

- Load the doubleword at address *xs1*
- Write the loaded value into *xd*
- XOR the value of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.72.3. Access

M
Always

B.72.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.72.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<64>(virtual_address, X[xs2], AmoOperation::Xor, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.72.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
            };
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.72.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

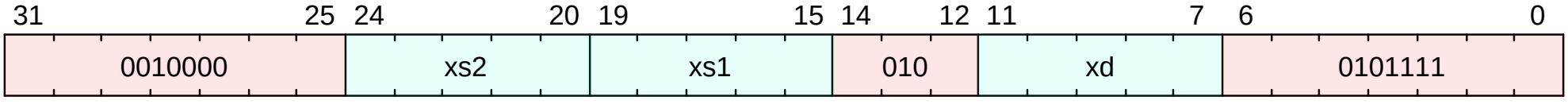
B.73. amoxor.w

Atomic fetch-and-xor word

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.73.1. Encoding



B.73.2. Description

Atomically:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- XOR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.73.3. Access

M
Always

B.73.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.73.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Xor, 1'b0, 1'b0, $encoding);
```

B.73.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
            };
```

```
            let rs2_val : xlenbits = match width {
              BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
              HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
              WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
              DOUBLE => X(rs2)
            };
            match (eares) {
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(_) => {
                let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
                  (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
                  (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
                  (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (mval) {
                  MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(loaded) => {
                    let result : xlenbits =
                      match op {
                        AMOSWAP => rs2_val,
                        AMOADD  => rs2_val + loaded,
                        AMOXOR  => rs2_val ^ loaded,
                        AMOAND  => rs2_val & loaded,
                        AMOOR   => rs2_val | loaded,

                        /* These operations convert bitvectors to integer values using [un]signed,
                         * and back using to_bits().
                         */
                        AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                        AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                        AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                        AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                      };
                    let rval : xlenbits = match width {
                      BYTE   => sign_extend(loaded[7..0]),
                      HALF   => sign_extend(loaded[15..0]),
                      WORD   => sign_extend(loaded[31..0]),
                      DOUBLE => loaded
                    };
                    let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                    };
                    match (wval) {
                      MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                      MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            } else {
              handle_illegal();
              RETIRE_FAIL
            }
          }
```

B.73.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

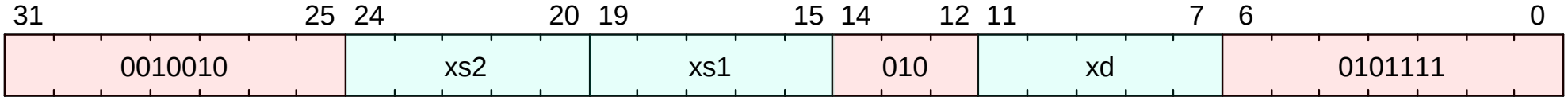
B.74. amoxor.w.aq

Atomic fetch-and-xor word (acquire)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.74.1. Encoding



B.74.2. Description

Atomically with acquire ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- XOR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.74.3. Access

M
Always

B.74.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.74.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Xor, 1'b1, 1'b0, $encoding);
```

B.74.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.74.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

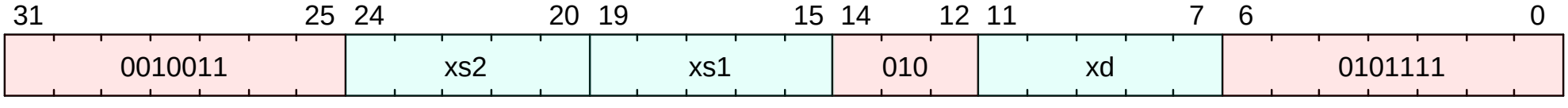
B.75. amoxor.w.aqrl

Atomic fetch-and-xor word (acquire-release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.75.1. Encoding



B.75.2. Description

Atomically with acquire and release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- XOR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.75.3. Access

M
Always

B.75.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.75.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
memory_model_acquire();
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Xor, 1'b1, 1'b1, $encoding);
memory_model_release();
```

B.75.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
```

```
        _       => false
      };
      let rs2_val : xlenbits = match width {
        BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
        HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
        WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
        DOUBLE => X(rs2)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
            (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
            (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
            (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
            (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
            _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
          };
          match (mval) {
            MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(loaded) => {
              let result : xlenbits =
                match op {
                  AMOSWAP => rs2_val,
                  AMOADD  => rs2_val + loaded,
                  AMOXOR  => rs2_val ^ loaded,
                  AMOAND  => rs2_val & loaded,
                  AMOOR   => rs2_val | loaded,

                  /* These operations convert bitvectors to integer values using [un]signed,
                   * and back using to_bits().
                   */
                  AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                  AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                  AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                  AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                };
              let rval : xlenbits = match width {
                BYTE   => sign_extend(loaded[7..0]),
                HALF   => sign_extend(loaded[15..0]),
                WORD   => sign_extend(loaded[31..0]),
                DOUBLE => loaded
              };
              let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
              };
              match (wval) {
                MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      } else {
        handle_illegal();
        RETIRE_FAIL
      }
    }
```

B.75.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

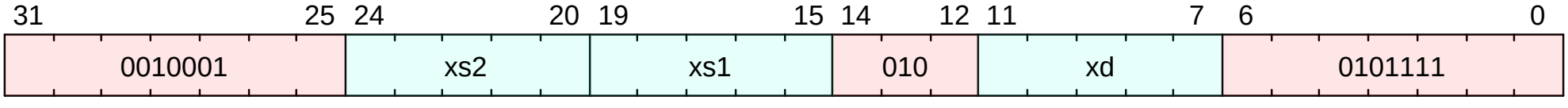
B.76. amoxor.w.rl

Atomic fetch-and-xor word (release)

This instruction is defined by:

Zaamo, version >= Zaamo@1.0.0

B.76.1. Encoding



B.76.2. Description

Atomically with release ordering:

- Load the word at address *xs1*
- Write the sign-extended value into *xd*
- XOR the least-significant word of register *xs2* to the loaded value
- Write the result to the address in *xs1*

B.76.3. Access

M
Always

B.76.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.76.5. IDL Operation

```
if (!implemented?(ExtensionName::A) || (MISA_CSR_IMPLEMENTED && (misa.A == 1'b0))) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
X[xd] = amo<32>(virtual_address, X[xs2][31:0], AmoOperation::Xor, 1'b0, 1'b1, $encoding);
memory_model_release();
```

B.76.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Some extensions perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), ReadWrite(Data, Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        match translateAddr(vaddr, ReadWrite(Data, Data)) {
          TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          TR_Address(addr, _) => {
            let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
              (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
              (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            let is_unsigned : bool = match op {
              AMOMINU => true,
              AMOMAXU => true,
              _       => false
```

```
        };
        let rs2_val : xlenbits = match width {
          BYTE   => if is_unsigned then zero_extend(X(rs2)[7..0])  else sign_extend(X(rs2)[7..0]),
          HALF   => if is_unsigned then zero_extend(X(rs2)[15..0]) else sign_extend(X(rs2)[15..0]),
          WORD   => if is_unsigned then zero_extend(X(rs2)[31..0]) else sign_extend(X(rs2)[31..0]),
          DOUBLE => X(rs2)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let mval : MemoryOpResult(xlenbits) = match (width, sizeof(xlen)) {
              (BYTE, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 1, aq, aq & rl, true)),
              (HALF, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 2, aq, aq & rl, true)),
              (WORD, _)    => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 4, aq, aq & rl, true)),
              (DOUBLE, 64) => extend_value(is_unsigned, mem_read(ReadWrite(Data, Data), addr, 8, aq, aq & rl, true)),
              _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
            };
            match (mval) {
              MemException(e)  => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              MemValue(loaded) => {
                let result : xlenbits =
                  match op {
                    AMOSWAP => rs2_val,
                    AMOADD  => rs2_val + loaded,
                    AMOXOR  => rs2_val ^ loaded,
                    AMOAND  => rs2_val & loaded,
                    AMOOR   => rs2_val | loaded,

                    /* These operations convert bitvectors to integer values using [un]signed,
                     * and back using to_bits().
                     */
                    AMOMIN  => to_bits(sizeof(xlen), min(signed(rs2_val),   signed(loaded))),
                    AMOMAX  => to_bits(sizeof(xlen), max(signed(rs2_val),   signed(loaded))),
                    AMOMINU => to_bits(sizeof(xlen), min(unsigned(rs2_val), unsigned(loaded))),
                    AMOMAXU => to_bits(sizeof(xlen), max(unsigned(rs2_val), unsigned(loaded)))
                  };
                let rval : xlenbits = match width {
                  BYTE   => sign_extend(loaded[7..0]),
                  HALF   => sign_extend(loaded[15..0]),
                  WORD   => sign_extend(loaded[31..0]),
                  DOUBLE => loaded
                };
                let wval : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_value(addr, 1, result[7..0],  aq & rl, rl, true),
                  (HALF, _)    => mem_write_value(addr, 2, result[15..0], aq & rl, rl, true),
                  (WORD, _)    => mem_write_value(addr, 4, result[31..0], aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_value(addr, 8, result,        aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                };
                match (wval) {
                  MemValue(true)  => { X(rd) = rval; RETIRE_SUCCESS },
                  MemValue(false) => { internal_error(__FILE__, __LINE__, "AMO got false from mem_write_value") },
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                }
              }
            }
          }
        } else {
          handle_illegal();
          RETIRE_FAIL
        }
      }
```

B.76.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

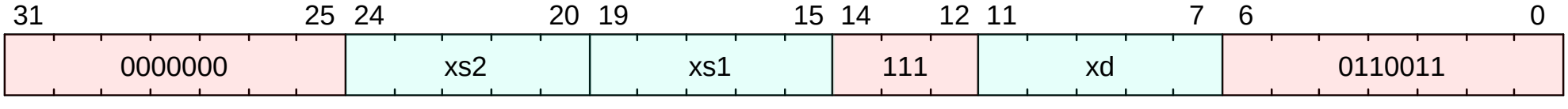
B.77. and

And

This instruction is defined by:

I, version >= I@2.1.0

B.77.1. Encoding



B.77.2. Description

And xs1 with xs2, and store the result in xd

B.77.3. Access

M
Always

B.77.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.77.5. IDL Operation

```
X[xd] = X[xs1] & X[xs2];
```

B.77.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.77.7. Exceptions

This instruction does not generate synchronous exceptions.

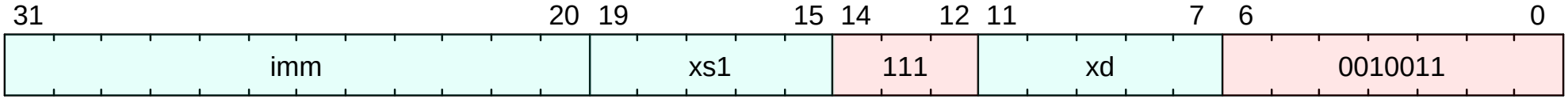
B.78. andi

And immediate

This instruction is defined by:

I, version >= I@2.1.0

B.78.1. Encoding



B.78.2. Description

And an immediate to the value in xs1, and store the result in xd

B.78.3. Access

M
Always

B.78.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.78.5. IDL Operation

```
X[xd] = X[xs1] & $signed(imm);
```

B.78.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => xs1_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(xs1_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(xs1_val <_u immext)),
    RISCV_ANDI  => xs1_val & immext,
    RISCV_ORI   => xs1_val | immext,
    RISCV_XORI  => xs1_val ^ immext
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.78.7. Exceptions

This instruction does not generate synchronous exceptions.

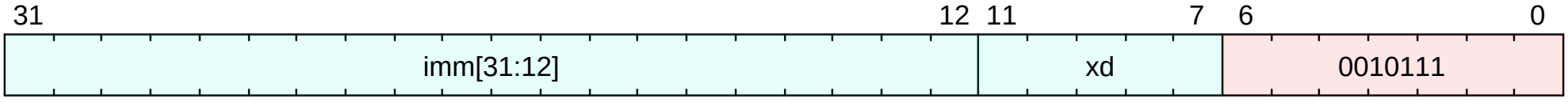
B.79. auipc

Add upper immediate to pc

This instruction is defined by:

I, version >= I@2.1.0

B.79.1. Encoding



B.79.2. Description

Add an immediate to the current PC.

B.79.3. Access

M
Always

B.79.4. Decode Variables

```
Bits<32> imm = {$encoding[31:12], 12'd0};
Bits<5> xd = $encoding[11:7];
```

B.79.5. IDL Operation

```
X[xd] = $pc + $signed(imm);
```

B.79.6. Sail Operation

```
{
  let off : xlenbits = sign_extend(imm @ 0x000);
  let ret : xlenbits = match op {
    RISCV_LUI   => off,
    RISCV_AUIPC => get_arch_pc() + off
  };
  X(xd) = ret;
  RETIRE_SUCCESS
}
```

B.79.7. Exceptions

This instruction does not generate synchronous exceptions.

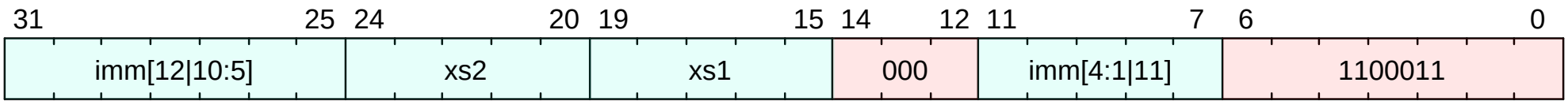
B.80. beq

Branch if equal

This instruction is defined by:

I, version >= I@2.1.0

B.80.1. Encoding



B.80.2. Description

Branch to PC + imm if the value in register xs1 is equal to the value in register xs2.

Raise a `MisalignedAddress` exception if PC + imm is misaligned.

B.80.3. Access

M
Always

B.80.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.80.5. IDL Operation

```
XReg lhs = X[xs1];
XReg rhs = X[xs2];
if (lhs == rhs) {
  jump_halfword($pc + $signed(imm));
}
```

B.80.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISCV_BEQ  => xs1_val == xs2_val,
    RISCV_BNE  => xs1_val != xs2_val,
    RISCV_BLT  => xs1_val <_s xs2_val,
    RISCV_BGE  => xs1_val >=_s xs2_val,
    RISCV_BLTU => xs1_val <_u xs2_val,
    RISCV_BGEU => xs1_val >=_u xs2_val
  };
  let t : xlenbits = PC + sign_extend(imm);
  if taken then {
    /* Extensions get the first checks on the prospective target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & not(extension("C")) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
```

```
    }
  } else RETIRE_SUCCESS
}
```

B.80.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

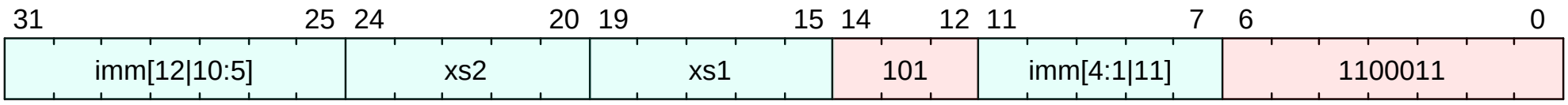
B.81. bge

Branch if greater than or equal

This instruction is defined by:

I, version >= I@2.1.0

B.81.1. Encoding



B.81.2. Description

Branch to PC + imm if the signed value in register xs1 is greater than or equal to the signed value in register xs2.

Raise a `MisalignedAddress` exception if PC + imm is misaligned.

B.81.3. Access

M
Always

B.81.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.81.5. IDL Operation

```
XReg lhs = X[xs1];
XReg rhs = X[xs2];
if ($signed(lhs) >= $signed(rhs)) {
  jump_halfword($pc + $signed(imm));
}
```

B.81.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISCV_BEQ  => xs1_val == xs2_val,
    RISCV_BNE  => xs1_val != xs2_val,
    RISCV_BLT  => xs1_val <_s xs2_val,
    RISCV_BGE  => xs1_val >=_s xs2_val,
    RISCV_BLTU => xs1_val <_u xs2_val,
    RISCV_BGEU => xs1_val >=_u xs2_val
  };
  let t : xlenbits = PC + sign_extend(imm);
  if taken then {
    /* Extensions get the first checks on the prospective target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & not(extension("C")) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
```

```
    }
  } else RETIRE_SUCCESS
}
```

B.81.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

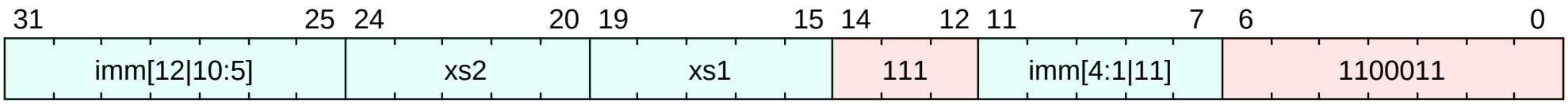
B.82. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.82.1. Encoding



B.82.2. Description

Branch to PC + imm if the unsigned value in register xs1 is greater than or equal to the unsigned value in register xs2.

Raise a `MisalignedAddress` exception if PC + imm is misaligned.

B.82.3. Access

M
Always

B.82.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.82.5. IDL Operation

```
XReg lhs = X[xs1];
XReg rhs = X[xs2];
if (lhs >= rhs) {
  jump_halfword($pc + $signed(imm));
}
```

B.82.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISCV_BEQ  => xs1_val == xs2_val,
    RISCV_BNE  => xs1_val != xs2_val,
    RISCV_BLT  => xs1_val <_s xs2_val,
    RISCV_BGE  => xs1_val >=_s xs2_val,
    RISCV_BLTU => xs1_val <_u xs2_val,
    RISCV_BGEU => xs1_val >=_u xs2_val
  };
  let t : xlenbits = PC + sign_extend(imm);
  if taken then {
    /* Extensions get the first checks on the prospective target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & not(extension("C")) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
```

```
    }
  } else RETIRE_SUCCESS
}
```

B.82.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

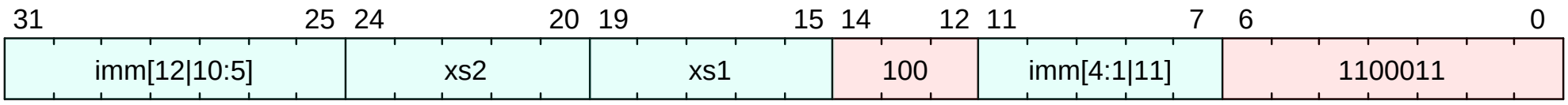
B.83. blt

Branch if less than

This instruction is defined by:

I, version >= I@2.1.0

B.83.1. Encoding



B.83.2. Description

Branch to PC + imm if the signed value in register xs1 is less than the signed value in register xs2.

Raise a `MisalignedAddress` exception if PC + imm is misaligned.

B.83.3. Access

M
Always

B.83.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.83.5. IDL Operation

```
XReg lhs = X[xs1];
XReg rhs = X[xs2];
if ($signed(lhs) < $signed(rhs)) {
  jump_halfword($pc + $signed(imm));
}
```

B.83.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISCV_BEQ  => xs1_val == xs2_val,
    RISCV_BNE  => xs1_val != xs2_val,
    RISCV_BLT  => xs1_val <_s xs2_val,
    RISCV_BGE  => xs1_val >=_s xs2_val,
    RISCV_BLTU => xs1_val <_u xs2_val,
    RISCV_BGEU => xs1_val >=_u xs2_val
  };
  let t : xlenbits = PC + sign_extend(imm);
  if taken then {
    /* Extensions get the first checks on the prospective target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & not(extension("C")) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
```

```
    }
  } else RETIRE_SUCCESS
}
```

B.83.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

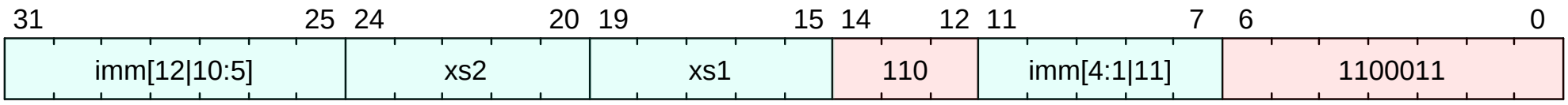
B.84. bltu

Branch if less than unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.84.1. Encoding



B.84.2. Description

Branch to PC + imm if the unsigned value in register xs1 is less than the unsigned value in register xs2.

Raise a `MisalignedAddress` exception if PC + imm is misaligned.

B.84.3. Access

M
Always

B.84.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.84.5. IDL Operation

```
XReg lhs = X[xs1];
XReg rhs = X[xs2];
if (lhs < rhs) {
  jump_halfword($pc + $signed(imm));
}
```

B.84.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISCV_BEQ  => xs1_val == xs2_val,
    RISCV_BNE  => xs1_val != xs2_val,
    RISCV_BLT  => xs1_val <_s xs2_val,
    RISCV_BGE  => xs1_val >=_s xs2_val,
    RISCV_BLTU => xs1_val <_u xs2_val,
    RISCV_BGEU => xs1_val >=_u xs2_val
  };
  let t : xlenbits = PC + sign_extend(imm);
  if taken then {
    /* Extensions get the first checks on the prospective target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & not(extension("C")) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
```

```
    }
  } else RETIRE_SUCCESS
}
```

B.84.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

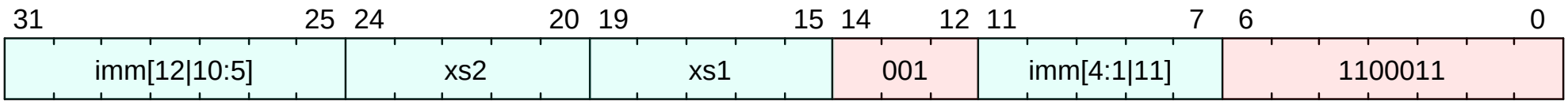
B.85. bne

Branch if not equal

This instruction is defined by:

I, version >= I@2.1.0

B.85.1. Encoding



B.85.2. Description

Branch to PC + imm if the value in register xs1 is not equal to the value in register xs2.

Raise a `MisalignedAddress` exception if PC + imm is misaligned.

B.85.3. Access

M
Always

B.85.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.85.5. IDL Operation

```
XReg lhs = X[xs1];
XReg rhs = X[xs2];
if (lhs != rhs) {
  jump_halfword($pc + $signed(imm));
}
```

B.85.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let taken : bool = match op {
    RISCV_BEQ  => xs1_val == xs2_val,
    RISCV_BNE  => xs1_val != xs2_val,
    RISCV_BLT  => xs1_val <_s xs2_val,
    RISCV_BGE  => xs1_val >=_s xs2_val,
    RISCV_BLTU => xs1_val <_u xs2_val,
    RISCV_BGEU => xs1_val >=_u xs2_val
  };
  let t : xlenbits = PC + sign_extend(imm);
  if taken then {
    /* Extensions get the first checks on the prospective target address. */
    match ext_control_check_pc(t) {
      Ext_ControlAddr_Error(e) => {
        ext_handle_control_check_error(e);
        RETIRE_FAIL
      },
      Ext_ControlAddr_OK(target) => {
        if bit_to_bool(target[1]) & not(extension("C")) then {
          handle_mem_exception(target, E_Fetch_Addr_Align());
          RETIRE_FAIL;
        } else {
          set_next_pc(target);
          RETIRE_SUCCESS
        }
      }
```

```
    }
  } else RETIRE_SUCCESS
}
```

B.85.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

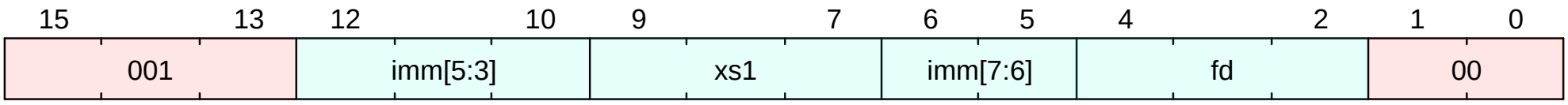
B.86. c.fld

Load double-precision

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** D, version >= D@2.2.0 * Zcd, version >= Zcd@1.0.0

B.86.1. Encoding



B.86.2. Description

Loads a double precision floating-point value from memory into register fd. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register xs1. It expands to fld `fd, offset(xs1)`.

B.86.3. Access

M
Always

B.86.4. Decode Variables

```
Bits<8> imm = {$encoding[6:5], $encoding[12:10], 3'd0};
Bits<3> fd = $encoding[4:2];
Bits<3> xs1 = $encoding[9:7];
```

B.86.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(xs1)] + imm;
f[fd] = sext(read_memory<64>(virtual_address, $encoding), 64);
```

B.86.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

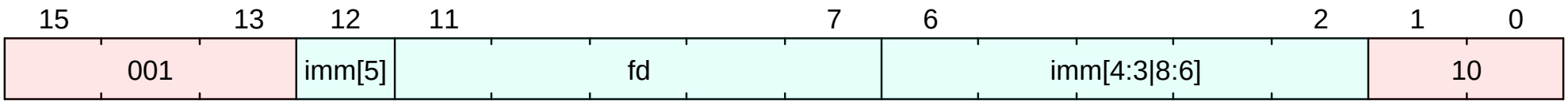
B.87. c.fldsp

Load doubleword into floating-point register from stack

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** D, version >= D@2.2.0 * Zcd, version >= Zcd@1.0.0

B.87.1. Encoding



B.87.2. Description

Loads a double-precision floating-point value from memory into floating-point register fd. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to fld `fd, offset(x2)`.

B.87.3. Access

M
Always

B.87.4. Decode Variables

```
Bits<9> imm = {$encoding[4:2], $encoding[12], $encoding[6:5], 3'd0};
Bits<5> fd = $encoding[11:7];
```

B.87.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (implemented?(ExtensionName::D) && (misa.D == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
f[fd] = read_memory<64>(virtual_address, $encoding);
```

B.87.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

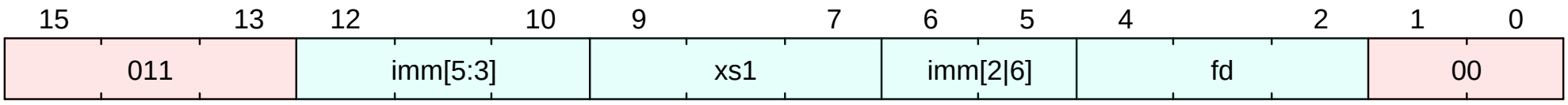
B.88. c.flw

Load single-precision

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** F, version >= F@2.2.0 * Zcf, version >= Zcf@1.0.0

B.88.1. Encoding



B.88.2. Description

Loads a single precision floating-point value from memory into register fd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register xs1. It expands to flw `fd, offset(xs1)`.

B.88.3. Access

M
Always

B.88.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> fd = $encoding[4:2];
Bits<3> xs1 = $encoding[9:7];
```

B.88.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(xs1)] + imm;
X[creg2reg(fd)] = sext(read_memory<32>(virtual_address, $encoding), 32);
```

B.88.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

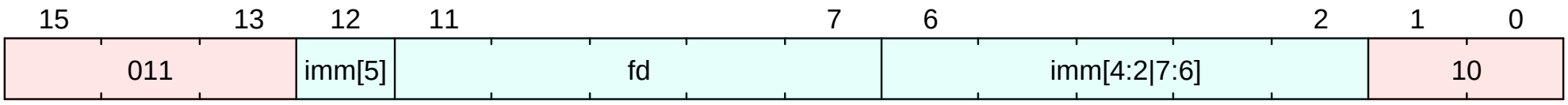
B.89. c.flwsp

Load word into floating-point register from stack

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** F, version >= F@2.2.0 * Zcf, version >= Zcf@1.0.0

B.89.1. Encoding



B.89.2. Description

Loads a single-precision floating-point value from memory into floating-point register fd. It computes its effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to flw `fd, offset(x2)`.

B.89.3. Access

M
Always

B.89.4. Decode Variables

```
Bits<8> imm = {$encoding[3:2], $encoding[12], $encoding[6:4], 2'd0};
Bits<5> fd = $encoding[11:7];
```

B.89.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (implemented?(ExtensionName::F) && (misa.F == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
f[fd] = read_memory<32>(virtual_address, $encoding);
```

B.89.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

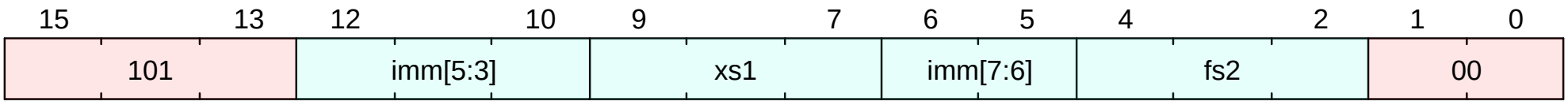
B.90. c.fsd

Store double-precision

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** D, version >= D@2.2.0 * Zcd, version >= Zcd@1.0.0

B.90.1. Encoding



B.90.2. Description

Stores a double precision floating-point value in register fs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register xs1. It expands to fsd `fs2, offset(xs1)`.

B.90.3. Access

M
Always

B.90.4. Decode Variables

```
Bits<8> imm = {$encoding[6:5], $encoding[12:10], 3'd0};
Bits<3> fs2 = $encoding[4:2];
Bits<3> xs1 = $encoding[9:7];
```

B.90.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(xs1)] + imm;
write_memory<64>(virtual_address, X[creg2reg(fs2)], $encoding);
```

B.90.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

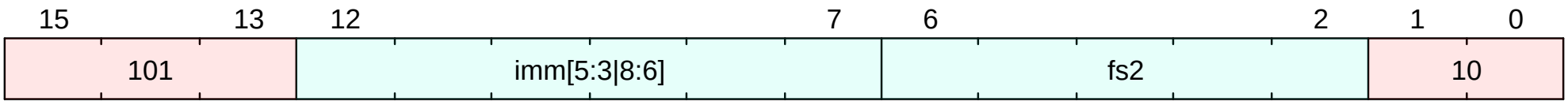
B.91. c.fsdsp

Store double-precision value to stack

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** D, version >= D@2.2.0 * Zcd, version >= Zcd@1.0.0

B.91.1. Encoding



B.91.2. Description

Stores a double-precision floating-point value in floating-point register fs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to fsd `fs2, offset(x2)`.

B.91.3. Access

M
Always

B.91.4. Decode Variables

```
Bits<9> imm = {$encoding[9:7], $encoding[12:10], 3'd0};
Bits<5> fs2 = $encoding[6:2];
```

B.91.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (implemented?(ExtensionName::D) && (misa.D == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<64>(virtual_address, f[fs2][63:0], $encoding);
```

B.91.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

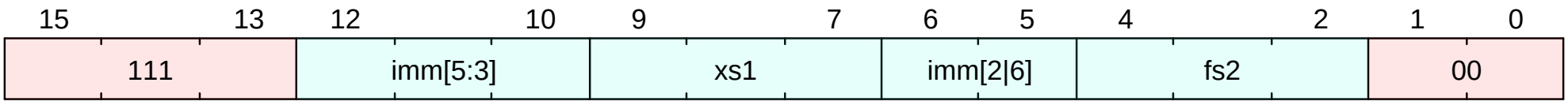
B.92. c.fsw

Store single-precision

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** F, version >= F@2.2.0 * Zcf, version >= Zcf@1.0.0

B.92.1. Encoding



B.92.2. Description

Stores a single precision floating-point value in register fs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register xs1. It expands to fsw `fs2, offset(xs1)`.

B.92.3. Access

M
Always

B.92.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> fs2 = $encoding[4:2];
Bits<3> xs1 = $encoding[9:7];
```

B.92.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(xs1)] + imm;
write_memory<32>(virtual_address, X[creg2reg(fs2)][31:0], $encoding);
```

B.92.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

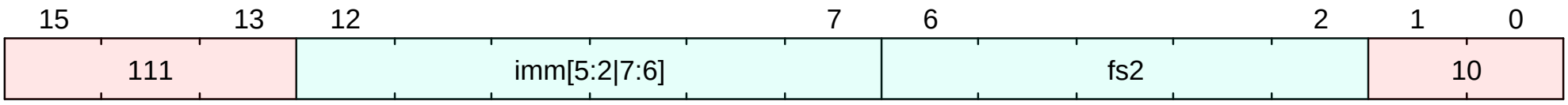
B.93. c.fswsp

Store single-precision value to stack

This instruction is defined by:

anyOf: * allOf: **C, version >= C@2.0.0** F, version >= F@2.2.0 * Zcf, version >= Zcf@1.0.0

B.93.1. Encoding



B.93.2. Description

Stores a single-precision floating-point value in floating-point register fs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to fsw `fs2, offset(x2)`.

B.93.3. Access

M
Always

B.93.4. Decode Variables

```
Bits<8> imm = {$encoding[8:7], $encoding[12:9], 2'd0};
Bits<5> fs2 = $encoding[6:2];
```

B.93.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (implemented?(ExtensionName::F) && (misa.F == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<32>(virtual_address, f[fs2][31:0], $encoding);
```

B.93.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

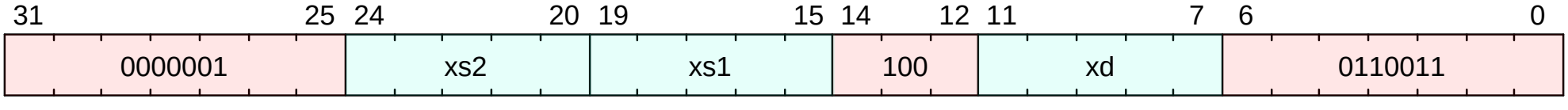
B.94. div

Signed division

This instruction is defined by:

M, version >= M@2.0.0

B.94.1. Encoding



B.94.2. Description

Divide xs1 by xs2, and store the result in xd. The remainder is discarded.

Division by zero will put -1 into xd.

Division resulting in signed overflow (when most negative number is divided by -1) will put the most negative number into xd;

B.94.3. Access

M
Always

B.94.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.94.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[xs1];
XReg src2 = X[xs2];
XReg signed_min = (xlen() == 32) ? $signed({1'b1, {31{1'b0}}}) : {1'b1, {63{1'b0}}};
if (src2 == 0) {
  X[xd] = {MXLEN{1'b1}};
} else if ((src1 == signed_min) && (src2 == {MXLEN{1'b1}})) {
  X[xd] = signed_min;
} else {
  X[xd] = $signed(src1) / $signed(src2);
}
```

B.94.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let q : int = if rs2_int == 0 then -1 else quot_round_zero(rs1_int, rs2_int);
    /* check for signed overflow */
    let q': int = if s & q > xlen_max_signed then xlen_min_signed else q;
    X(rd) = to_bits(sizeof(xlen), q');
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.94.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

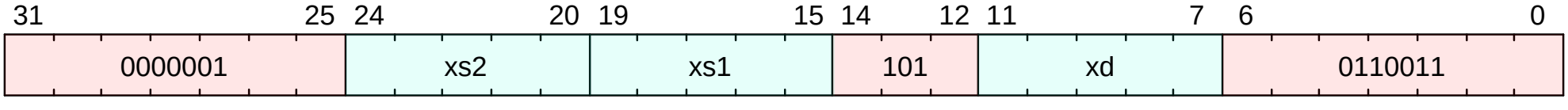
B.95. divu

Unsigned division

This instruction is defined by:

M, version >= M@2.0.0

B.95.1. Encoding



B.95.2. Description

Divide unsigned values in xs1 by xs2, and store the result in xd.

The remainder is discarded.

If the value in xs2 is zero, xd gets the largest unsigned value.

B.95.3. Access

M
Always

B.95.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.95.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[xs1];
XReg src2 = X[xs2];
if (src2 == 0) {
  X[xd] = {MXLEN{1'b1}};
} else {
  X[xd] = src1 / src2;
}
```

B.95.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let q : int = if rs2_int == 0 then -1 else quot_round_zero(rs1_int, rs2_int);
    /* check for signed overflow */
    let q': int = if s & q > xlen_max_signed then xlen_min_signed else q;
    X(rd) = to_bits(sizeof(xlen), q');
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.95.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

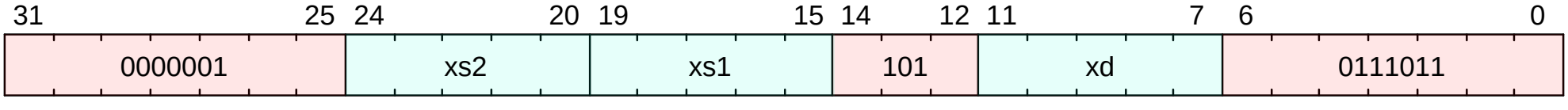
B.96. divuw

Unsigned 32-bit division

This instruction is defined by:

M, version >= M@2.0.0

B.96.1. Encoding



B.96.2. Description

Divide the unsigned 32-bit values in xs1 and xs2, and store the sign-extended result in xd.

The remainder is discarded.

If the value in xs2 is zero, xd is written with all 1s.

B.96.3. Access

M
Always

B.96.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.96.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[xs1][31:0];
Bits<32> src2 = X[xs2][31:0];
if (src2 == 0) {
  X[xd] = {64{1'b1}};
} else {
  Bits<32> result = src1 / src2;
  Bits<1> sign_bit = result[31];
  X[xd] = {{32{sign_bit}}, result};
}
```

B.96.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let q : int = if rs2_int == 0 then -1 else quot_round_zero(rs1_int, rs2_int);
    /* check for signed overflow */
    let q': int = if s & q > (2 ^ 31 - 1) then  (0 - 2^31) else q;
    X(rd) = sign_extend(to_bits(32, q'));
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.96.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

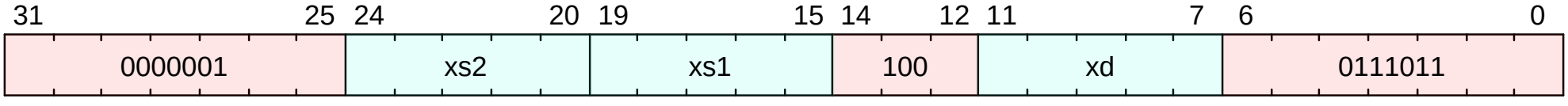
B.97. divw

Signed 32-bit division

This instruction is defined by:

M, version >= M@2.0.0

B.97.1. Encoding



B.97.2. Description

Divide the lower 32-bits of register xs1 by the lower 32-bits of register xs2, and store the sign-extended result in xd.

The remainder is discarded.

Division by zero will put -1 into xd.

Division resulting in signed overflow (when most negative number is divided by -1) will put the most negative number into xd;

B.97.3. Access

M
Always

B.97.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.97.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[xs1][31:0];
Bits<32> src2 = X[xs2][31:0];
if (src2 == 0) {
  X[xd] = {MXLEN{1'b1}};
} else if ((src1 == {33'b1, 31'b0}) && (src2 == 32'b1)) {
  X[xd] = {33'b1, 31'b0};
} else {
  Bits<32> result = $signed(src1) / $signed(src2);
  Bits<1> sign_bit = result[31];
  X[xd] = {{32{sign_bit}}, result};
}
```

B.97.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let q : int = if rs2_int == 0 then -1 else quot_round_zero(rs1_int, rs2_int);
    /* check for signed overflow */
    let q': int = if s & q > (2 ^ 31 - 1) then  (0 - 2^31) else q;
    X(rd) = sign_extend(to_bits(32, q'));
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
```

```
}
```

B.97.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

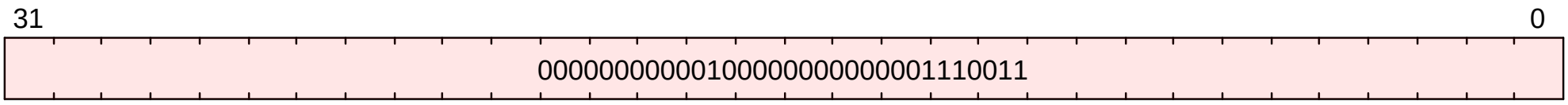
B.98. ebreak

Breakpoint exception

This instruction is defined by:

I, version >= I@2.1.0

B.98.1. Encoding



B.98.2. Description

The EBREAK instruction is used by debuggers to cause control to be transferred back to a debugging environment. Unless overridden by an external debug environment, EBREAK raises a breakpoint exception and performs no other operation.

As described in the C Standaxd Extension for Compressed Instructions, the c.ebreak instruction performs the same operation as the EBREAK instruction.

EBREAK causes the receiving privilege mode's epc register to be set to the address of the EBREAK instruction itself, not the address of the following instruction. As EBREAK causes a synchronous exception, it is not considered to retire, and should not increment the minstret CSR.

B.98.3. Access

M
Always

B.98.4. Decode Variables

```
```

B.98.5. IDL Operation

```
if (TRAP_ON_EBREAK) {
  raise_precise(ExceptionCode::Breakpoint, mode(), $pc);
} else {
  eei_ebreak();
}
```

B.98.6. Sail Operation

```
{
  handle_mem_exception(PC, E_Breakpoint());
  RETIRE_FAIL
}
```

B.98.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

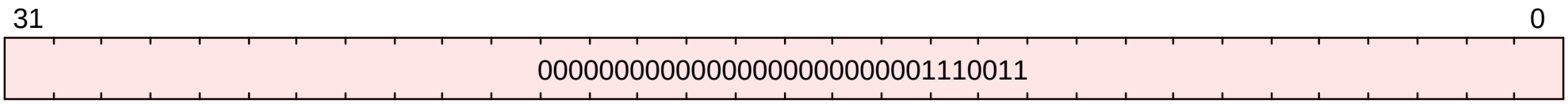
B.99. ecall

Environment call

This instruction is defined by:

I, version >= I@2.1.0

B.99.1. Encoding



B.99.2. Description

Makes a request to the supporting execution environment. When executed in U-mode, S-mode, or M-mode, it generates an environment-call-from-U-mode exception, environment-call-from-S-mode exception, or environment-call-from-M-mode exception, respectively, and performs no other operation.

ECALL generates a different exception for each originating privilege mode so that environment call exceptions can be selectively delegated. A typical use case for Unix-like operating systems is to delegate to S-mode the environment-call-from-U-mode exception but not the others.

ECALL causes the receiving privilege mode's epc register to be set to the address of the ECALL instruction itself, not the address of the following instruction. As ECALL causes a synchronous exception, it is not considered to retire, and should not increment the minstret CSR.

B.99.3. Access

M
Always

B.99.4. Decode Variables

```
```

B.99.5. IDL Operation

```
if (mode() == PrivilegeMode::M) {
  if (TRAP_ON_ECALL_FROM_M) {
    raise_precise(ExceptionCode::Mcall, PrivilegeMode::M, 0);
  } else {
    eei_ecall_from_m();
  }
} else if (mode() == PrivilegeMode::S) {
  if (TRAP_ON_ECALL_FROM_S) {
    raise_precise(ExceptionCode::Scall, PrivilegeMode::S, 0);
  } else {
    eei_ecall_from_s();
  }
} else if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::VU) {
  if (TRAP_ON_ECALL_FROM_U) {
    raise_precise(ExceptionCode::Ucall, mode(), 0);
  } else {
    eei_ecall_from_u();
  }
} else if (mode() == PrivilegeMode::VS) {
  if (TRAP_ON_ECALL_FROM_VS) {
    raise_precise(ExceptionCode::VScall, PrivilegeMode::VS, 0);
  } else {
    eei_ecall_from_vs();
  }
}
```

B.99.6. Sail Operation

```
{
  let t : sync_exception =
    struct { trap = match (cur_privilege) {
```

```
                    User       => E_U_EnvCall(),
                    Supervisor => E_S_EnvCall(),
                    Machine    => E_M_EnvCall()
                  },
            excinfo = (None() : option(xlenbits)),
            ext     = None() };
  set_next_pc(exception_handler(cur_privilege, CTL_TRAP(t), PC));
  RETIRE_FAIL
}
```

B.99.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

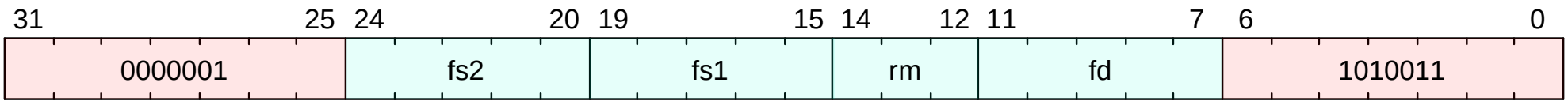
B.100. fadd.d

Floating-Point Add Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.100.1. Encoding



B.100.2. Description

The fadd.d instruction is analogous to fadd.s and performs double-precision floating-point addition between `fs1` and `fs2` and writes the final result to `fd`.

B.100.3. Access

M
Always

B.100.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.100.5. IDL Operation

```

```

B.100.6. Exceptions

This instruction does not generate synchronous exceptions.

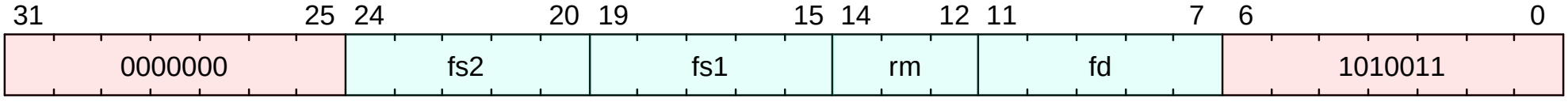
B.101. fadd.s

Floating-Point Add Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.101.1. Encoding



B.101.2. Description

The fadd.s instruction performs single-precision floating-point addition of `xs1` and `xs2` and writes the final result to `fd`.

B.101.3. Access

M
Always

B.101.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.101.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode mode = rm_to_mode(rm, $encoding);
f[fd] = f32_add(f[fs1], f[fs2], mode);
```

B.101.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) = match op {
        FADD_S  => riscv_f32Add (rm_3b, rs1_val_32b, rs2_val_32b),
        FSUB_S  => riscv_f32Sub (rm_3b, rs1_val_32b, rs2_val_32b),
        FMUL_S  => riscv_f32Mul (rm_3b, rs1_val_32b, rs2_val_32b),
        FDIV_S  => riscv_f32Div (rm_3b, rs1_val_32b, rs2_val_32b)
      };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.101.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

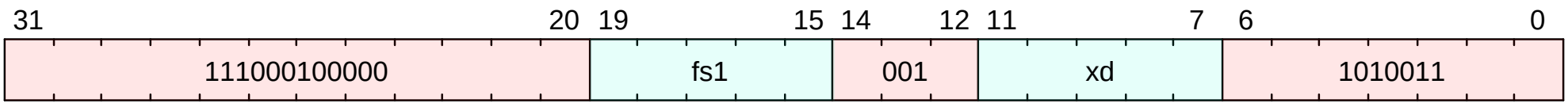
B.102. fclass.d

Floating-Point Classify Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.102.1. Encoding



B.102.2. Description

The fclass.d instruction is defined analogously to its single-precision counterpart, but operates on double-precision operands. It examines the value in floating-point register `fs1` and writes to integer register `xd` a 10-bit mask that indicates the class of the floating point number.

The format of the mask is described in the table below. The corresponding bit in `xd` will be set if the property is true and clear otherwise. All other bits in `xd` are cleared. Note that exactly one bit in `xd` will be set.

Table 9. Format of result of `fclass` instruction.

xd bit	Meaning
0	*fs1* is `-\infty`.
1	*fs1* is a negative normal number.
2	*fs1* is a negative subnormal number.
3	*fs1* is `-0`.
4	*fs1* is `+0`.
5	*fs1* is a positive subnormal number.
6	*fs1* is a positive normal number.
7	*fs1* is `+\infty`.
8	*fs1* is a signaling NaN.
9	*fs1* is a quiet NaN.

B.102.3. Access

M
Always

B.102.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.102.5. IDL Operation

```

```

B.102.6. Exceptions

This instruction does not generate synchronous exceptions.

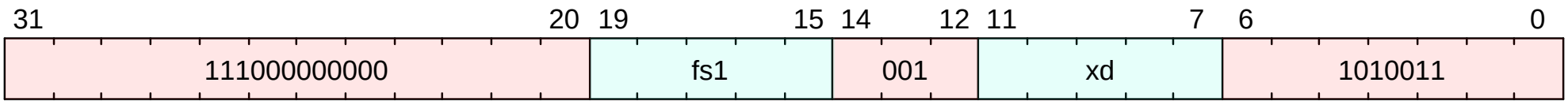
B.103. fclass.s

Floating-Point Classify Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.103.1. Encoding



B.103.2. Description

The fclass.s instruction examines the value in floating-point register `fs1` and writes to integer register `xd` a 10-bit mask that indicates the class of the floating-point number.

The format of the mask is described in the table below. The corresponding bit in `xd` will be set if the property is true and clear otherwise. All other bits in `xd` are cleared. Note that exactly one bit in `xd` will be set. fclass.s does not set the floating-point exception flags.

Table 10. Format of result of `fclass` instruction.

xd bit	Meaning
0	*fs1* is `-\infty`.
1	*fs1* is a negative normal number.
2	*fs1* is a negative subnormal number.
3	*fs1* is `-0`.
4	*fs1* is `+0`.
5	*fs1* is a positive subnormal number.
6	*fs1* is a positive normal number.
7	*fs1* is `+\infty`.
8	*fs1* is a signaling NaN.
9	*fs1* is a quiet NaN.

B.103.3. Access

M
Always

B.103.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.103.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = f[fs1][31:0];
if (is_sp_neg_inf?(sp_value)) {
  X[xd] = 1 << 0;
} else if (is_sp_neg_norm?(sp_value)) {
  X[xd] = 1 `<< 1;
} else if (is_sp_neg_subnorm?(sp_value)) {
  X[xd] = 1 `<< 2;
} else if (is_sp_neg_zero?(sp_value)) {
  X[xd] = 1 `<< 3;
} else if (is_sp_pos_zero?(sp_value)) {
  X[xd] = 1 `<< 4;
} else if (is_sp_pos_subnorm?(sp_value)) {
  X[xd] = 1 `<< 5;
} else if (is_sp_pos_norm?(sp_value)) {
  X[xd] = 1 `<< 6;
} else if (is_sp_pos_inf?(sp_value)) {
  X[xd] = 1 `<< 7;
```

```
} else if (is_sp_signaling_nan?(sp_value)) {
  X[xd] = 1 `<< 8;
} else {
  assert(is_sp_quiet_nan?(sp_value), "Unexpected SP value");
  X[xd] = 1 `<< 9;
}
```

B.103.6. Sail Operation

```
{
  let rs1_val_X            = X(rs1);
  let rd_val_S             = rs1_val_X [31..0];
  F(rd) = nan_box (rd_val_S);
  RETIRE_SUCCESS
}
```

B.103.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

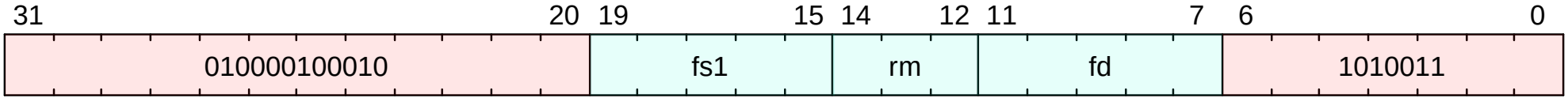
B.104. fcvt.d.h

No synopsis available

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfh, version >= Zfh@1.0.0

B.104.1. Encoding



B.104.2. Description

No description available.

B.104.3. Access

M
Always

B.104.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.104.5. IDL Operation

```

```

B.104.6. Exceptions

This instruction does not generate synchronous exceptions.

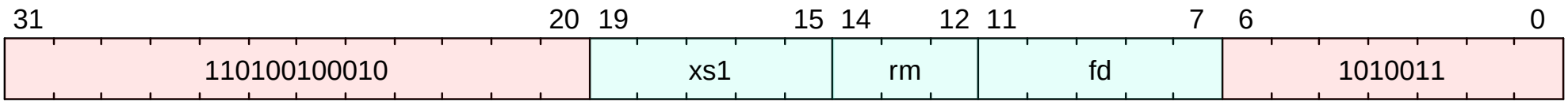
B.105. fcvt.d.l

Floating-Point Convert Long to Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.105.1. Encoding



B.105.2. Description

The fcvt.d.l instruction converts a 64-bit signed integer, in integer register `xs1` into a double-precision floating-point number in floating-point register `fd`.

B.105.3. Access

M
Always

B.105.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.105.5. IDL Operation

```

```

B.105.6. Exceptions

This instruction does not generate synchronous exceptions.

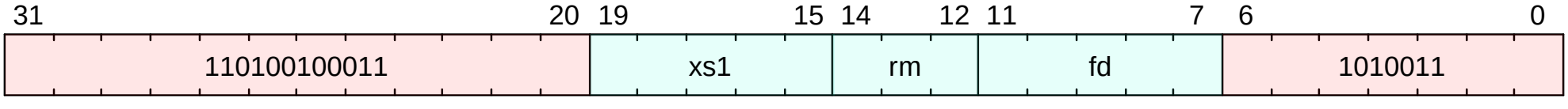
B.106. fcvt.d.lu

Floating-Point Convert Unsigned Long to Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.106.1. Encoding



B.106.2. Description

The fcvt.d.lu instruction converts to or from a 64-bit unsigned integer, `xs1` into a double-precision floating-point number in floating-point register `fd`.

B.106.3. Access

M
Always

B.106.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.106.5. IDL Operation

```

```

B.106.6. Exceptions

This instruction does not generate synchronous exceptions.

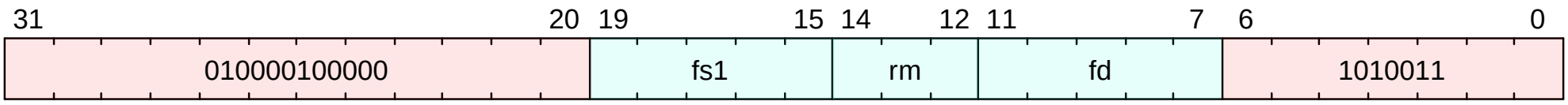
B.107. fcvt.d.s

Floating-Point Convert Single-Precision to Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.107.1. Encoding



B.107.2. Description

The single-precision to double-precision conversion instruction, fcvt.d.s is encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. The `xs2` field encodes the datatype of the source, and the `fmt` field encodes the datatype of the destination. fcvt.d.s will never round.

B.107.3. Access

M
Always

B.107.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.107.5. IDL Operation

```

```

B.107.6. Exceptions

This instruction does not generate synchronous exceptions.

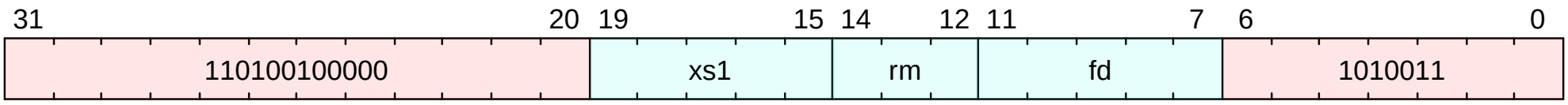
B.108. fcvt.d.w

Floating-Point Convert Word to Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.108.1. Encoding



B.108.2. Description

The fcvt.d.w instruction converts a 32-bit signed integer, in integer register `xs1` into a double-precision floating-point number in floating-point register `fd`. Note fcvt.d.w always produces an exact result and is unaffected by rounding mode.

B.108.3. Access

M
Always

B.108.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.108.5. IDL Operation

```

```

B.108.6. Exceptions

This instruction does not generate synchronous exceptions.

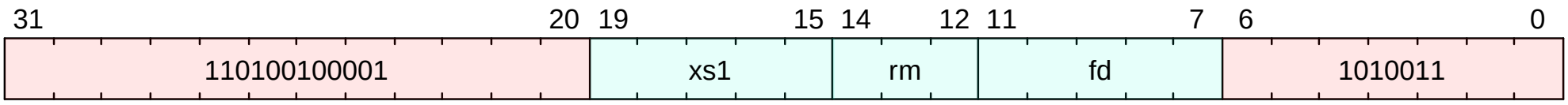
B.109. fcvt.d.wu

Floating-Point Convert Unsigned Word to Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.109.1. Encoding



B.109.2. Description

The fcvt.d.wu instruction converts a 32-bit unsigned integer in integer register `xs1` into a double-precision floating-point number in floating-point register `fd`. Note fcvt.d.wu always produces an exact result and is unaffected by rounding mode.

B.109.3. Access

M
Always

B.109.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.109.5. IDL Operation

```

```

B.109.6. Exceptions

This instruction does not generate synchronous exceptions.

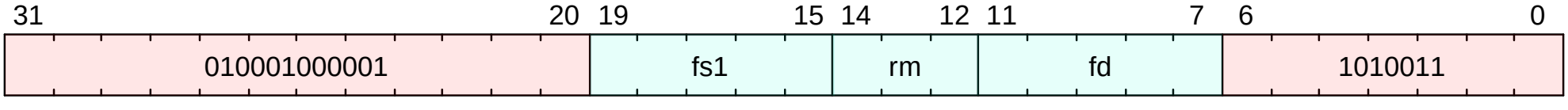
B.110. fcvt.h.d

Floating-point Convert Double-precision to Half-precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfh, version >= Zfh@1.0.0

B.110.1. Encoding



B.110.2. Description

fcvt.h.d converts a Double-precision Floating-point number to a Half-precision floating-point number.

B.110.3. Access

M
Always

B.110.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.110.5. IDL Operation

```

```

B.110.6. Exceptions

This instruction does not generate synchronous exceptions.

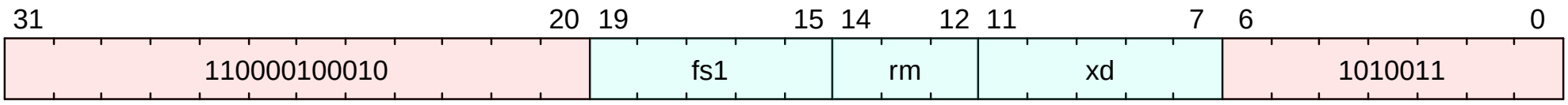
B.111. fcvt.l.d

Floating-Point Convert Double-Precision to Long

This instruction is defined by:

D, version >= D@2.2.0

B.111.1. Encoding



B.111.2. Description

The fcvt.l.d instruction converts a double-precision floating-point number in floating-point register `fs1` to a signed 64-bit integer, in integer register `xd`.

B.111.3. Access

M
Always

B.111.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.111.5. IDL Operation

```

```

B.111.6. Exceptions

This instruction does not generate synchronous exceptions.

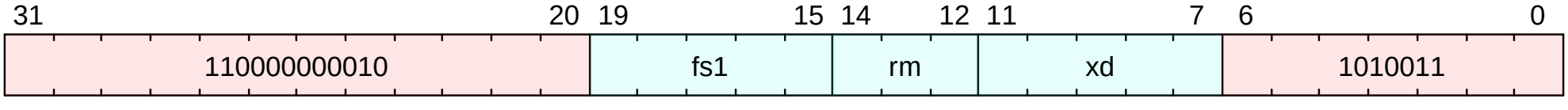
B.112. fcvt.l.s

Floating-Point Convert Single-Precision to Long

This instruction is defined by:

F, version >= F@2.2.0

B.112.1. Encoding



B.112.2. Description

The fcvt.l.s instruction converts a floating-point number in floating-point register `fs1` to a signed 64-bit integer, in integer register `xd`.

B.112.3. Access

M
Always

B.112.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.112.5. IDL Operation

```

```

B.112.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.112.7. Exceptions

This instruction does not generate synchronous exceptions.

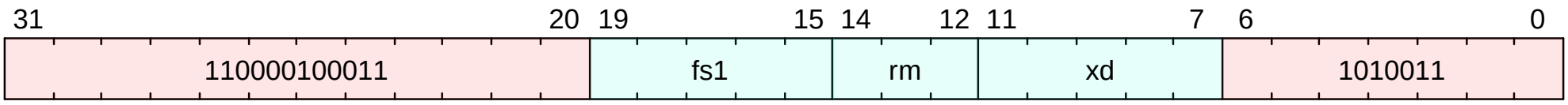
B.113. fcvt.lu.d

Floating-Point Convert Double-Precision to Unsigned Long

This instruction is defined by:

D, version >= D@2.2.0

B.113.1. Encoding



B.113.2. Description

The fcvt.lu.d instruction converts a double-precision floating-point number in floating-point register `fs1` to an unsigned 64-bit integer, in integer register `xd`.

B.113.3. Access

M
Always

B.113.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.113.5. IDL Operation

```

```

B.113.6. Exceptions

This instruction does not generate synchronous exceptions.

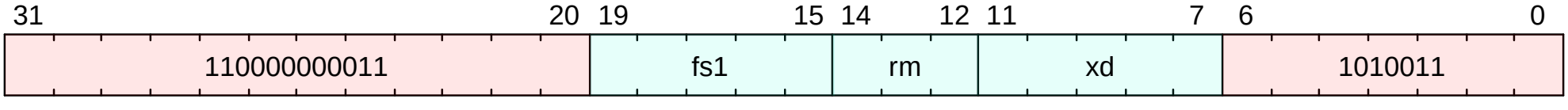
B.114. fcvt.lu.s

Floating-Point Convert Single-Precision to Unsigned Long

This instruction is defined by:

F, version >= F@2.2.0

B.114.1. Encoding



B.114.2. Description

The fcvt.lu.s instruction converts a floating-point number in floating-point register `fs1` to a unsigned 64-bit integer, in integer register `xd`.

B.114.3. Access

M
Always

B.114.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.114.5. IDL Operation

```

```

B.114.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.114.7. Exceptions

This instruction does not generate synchronous exceptions.

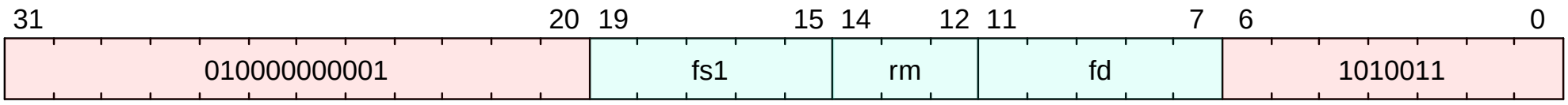
B.115. fcvt.s.d

Floating-Point Convert Double-Precision to Single-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.115.1. Encoding



B.115.2. Description

The fcvt.s.d instruction converts a double-precision floating-point number to a single-precision floating-point number. This is encoded in the OP-FP major opcode space and both the source and destination are floating-point registers. The `xs2` field encodes the datatype of the source, and the `fmt` field encodes the datatype of the destination. fcvt.s.d rounds according to the `rm` field.

B.115.3. Access

M
Always

B.115.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.115.5. IDL Operation

```

```

B.115.6. Exceptions

This instruction does not generate synchronous exceptions.

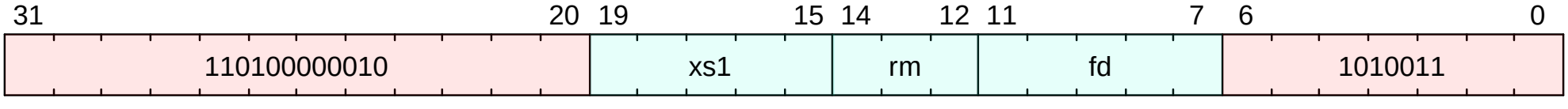
B.116. fcvt.s.l

Floating-Point Convert Long to Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.116.1. Encoding



B.116.2. Description

The fcvt.s.l instruction converts a 64-bit signed integer in integer register `xs1` into a floating-point number in floating-point register `fd`.

B.116.3. Access

M
Always

B.116.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.116.5. IDL Operation

```

```

B.116.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.116.7. Exceptions

This instruction does not generate synchronous exceptions.

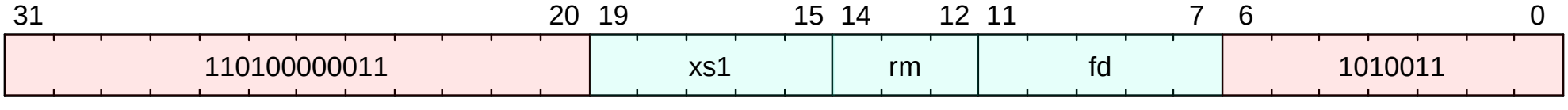
B.117. fcvt.s.lu

Floating-Point Convert Unsigned Long to Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.117.1. Encoding



B.117.2. Description

The fcvt.s.lu instruction converts a 64-bit unsigned integer into a single-precision floating-point number.

B.117.3. Access

M
Always

B.117.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.117.5. IDL Operation

```

```

B.117.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.117.7. Exceptions

This instruction does not generate synchronous exceptions.

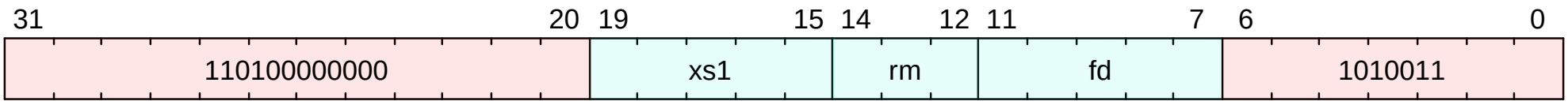
B.118. fcvt.s.w

Floating-Point Convert Word to Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.118.1. Encoding



B.118.2. Description

The fcvt.s.w instruction converts a 32-bit signed integer in integer register `xs1` into a floating-point number in floating-point register `fd`.

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w fd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

B.118.3. Access

M
Always

B.118.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.118.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
f[fd] = i32_to_f32(X[xs1], rounding_mode);
mark_f_state_dirty();
```

B.118.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.118.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

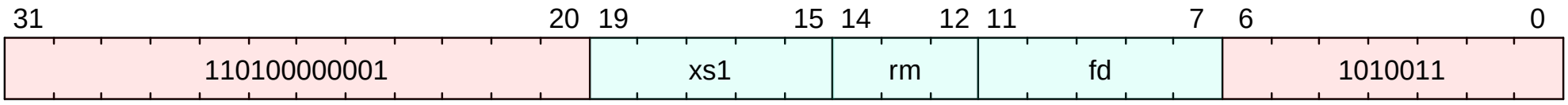
B.119. fcvt.s.wu

Floating-Point Convert Unsigned Word to Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.119.1. Encoding



B.119.2. Description

The fcvt.s.wu instruction converts a 32-bit unsigned integer in integer register `xs1` into a floating-point number in floating-point register `fd`.

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

B.119.3. Access

M
Always

B.119.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.119.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
f[fd] = ui32_to_f32(X[xs1], rounding_mode);
mark_f_state_dirty();
```

B.119.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.119.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

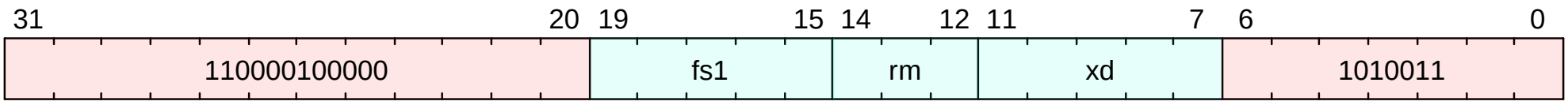
B.120. fcvt.w.d

Floating-Point Convert Double-Precision to Word

This instruction is defined by:

D, version >= D@2.2.0

B.120.1. Encoding



B.120.2. Description

The fcvt.w.d instruction converts a double-precision floating-point number in floating-point register `fs1` to a signed 32-bit integer, in integer register `xd`.

B.120.3. Access

M
Always

B.120.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.120.5. IDL Operation

```

```

B.120.6. Exceptions

This instruction does not generate synchronous exceptions.

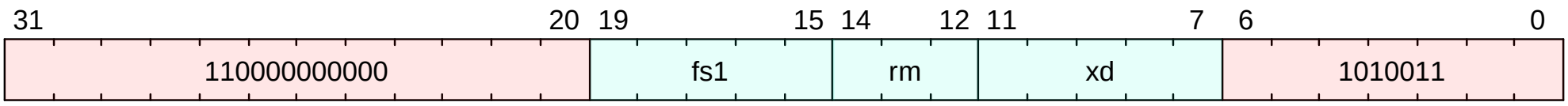
B.121. fcvt.w.s

Floating-Point Convert Single-Precision to Word

This instruction is defined by:

F, version >= F@2.2.0

B.121.1. Encoding



B.121.2. Description

The fcvt.w.s instruction converts a floating-point number in floating-point register `fs1` to a signed 32-bit integer in integer register `xd`. For XLEN >32, fcvt.w.s sign-extends the 32-bit result to the destination register width.

If the rounded result is not representable as a 32-bit signed integer, it is clipped to the nearest value and the invalid flag is set.

The range of valid inputs and behavior for invalid inputs are:

	Value
Minimum valid input (after rounding)	`-2^31`
Maximum valid input (after rounding)	`2^31 - 1`
Output for out-of-range negative input	`-2^31`
Output for <code>-∞</code>	`-2^31`
Output for out-of-range positive input	`2^31 - 1`
Output for <code>+∞</code> for <code>NaN</code>	`2^31 - 1`

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w xd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

B.121.3. Access

M
Always

B.121.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.121.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
X[xd] = f32_to_i32(f[fs1], rounding_mode);
```

B.121.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);
```

```
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.121.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

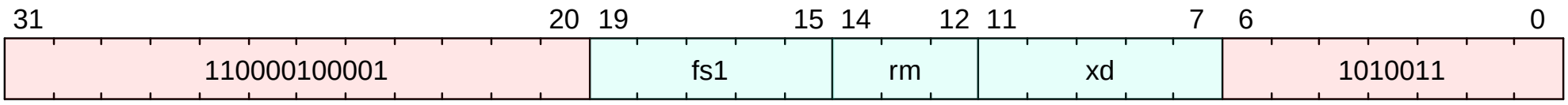
B.122. fcvt.wu.d

Floating-Point Convert Double-Precision to Unsigned Word

This instruction is defined by:

D, version >= D@2.2.0

B.122.1. Encoding



B.122.2. Description

The fcvt.wu.d instruction converts a double-precision floating-point number in floating-point register `fs1` to an unsigned 32-bit integer, in integer register `xd`.

B.122.3. Access

M
Always

B.122.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.122.5. IDL Operation

```

```

B.122.6. Exceptions

This instruction does not generate synchronous exceptions.

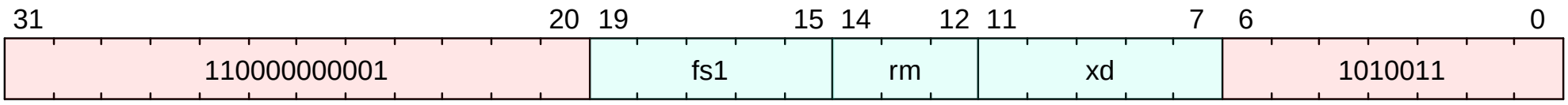
B.123. fcvt.wu.s

Floating-Point Convert Single-Precision to Unsigned Word

This instruction is defined by:

F, version >= F@2.2.0

B.123.1. Encoding



B.123.2. Description

Converts a floating-point number in floating-point register `fs1` to an unsigned 32-bit integer in integer register `xd`. For XLEN >32, fcvt.wu.s sign-extends the 32-bit result to the destination register width.

If the rounded result is not representable as a 32-bit unsigned integer, it is clipped to the nearest value and the invalid flag is set.

The range of valid inputs and behavior for invalid inputs are:

	Value
Minimum valid input (after rounding)	`0`
Maximum valid input (after rounding)	`2^32 - 1`
Output for out-of-range negative input	`0`
Output for <code>-∞</code>	`0`
Output for out-of-range positive input	`2^32 - 1`
Output for <code>+∞</code> for <code>NaN</code>	`2^32 - 1`

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w xd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

B.123.3. Access

M
Always

B.123.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.123.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
X[xd] = f32_to_ui32(f[fs1], rounding_mode);
```

B.123.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);
```

```
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.123.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

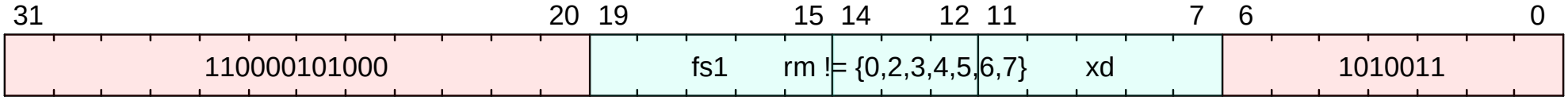
B.124. fcvtmod.w.d

Floating-Point Convert Double-Precision to Word with Modulo

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.124.1. Encoding



B.124.2. Description

The fcvtmod.w.d instruction always rounds towards zero. Bits 31:0 are taken from the rounded, unbounded two's complement result, then sign-extended to XLEN bits and written to integer register `xd`. ±∞ and NaN are converted to zero.

Floating-point exception flags are raised the same as they would be for fcvt.w.d with the same input operand.

This instruction is only provided if the D extension is implemented. It is encoded like fcvt.w.d, but with the `xs2` field set to 8 and the `rm` field set to 1 (RTZ). Other `rm` values are reserved.

B.124.3. Access

M
Always

B.124.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> xd = $encoding[11:7];
```

B.124.5. IDL Operation

```

```

B.124.6. Exceptions

This instruction does not generate synchronous exceptions.

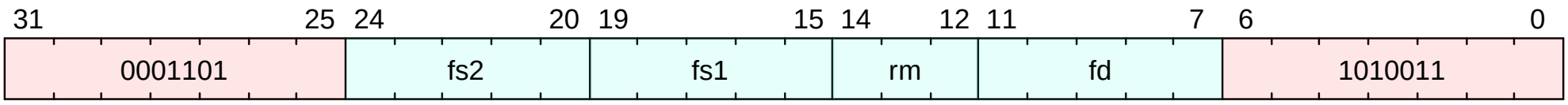
B.125. fdiv.d

Floating-Point Divide Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.125.1. Encoding



B.125.2. Description

The fdiv.d instruction performs the double-precision floating-point division of `fs1` by `fs2`. It is defined analogously to its single-precision counterpart, but operates on double-precision operands and produces double-precision results.

B.125.3. Access

M
Always

B.125.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.125.5. IDL Operation

```

```

B.125.6. Exceptions

This instruction does not generate synchronous exceptions.

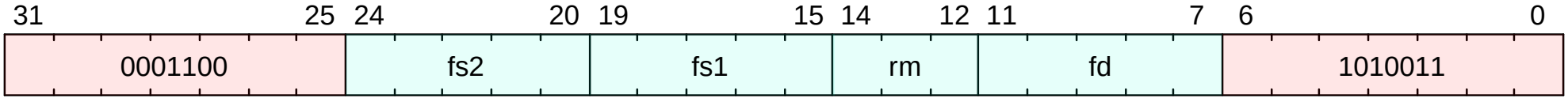
B.126. fdiv.s

Floating-Point Divide Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.126.1. Encoding



B.126.2. Description

The fdiv.s instruction performs the single-precision floating-point division of `fs1` by `fs2`, and writes the final result to `fd`.

B.126.3. Access

M
Always

B.126.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.126.5. IDL Operation

```

```

B.126.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) = match op {
        FADD_S  => riscv_f32Add (rm_3b, rs1_val_32b, rs2_val_32b),
        FSUB_S  => riscv_f32Sub (rm_3b, rs1_val_32b, rs2_val_32b),
        FMUL_S  => riscv_f32Mul (rm_3b, rs1_val_32b, rs2_val_32b),
        FDIV_S  => riscv_f32Div (rm_3b, rs1_val_32b, rs2_val_32b)
      };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.126.7. Exceptions

This instruction does not generate synchronous exceptions.

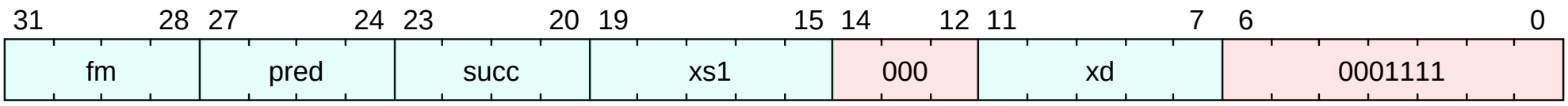
B.127. fence

Memory ordering fence

This instruction is defined by:

I, version >= I@2.1.0

B.127.1. Encoding



B.127.2. Description

Orders memory operations.

The fence instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a fence before any operation in the *predecessor* set preceding the fence.

The predecessor and successor fields have the same format to specify operation types:

`pred`				`succ`			
27	26	25	24	23	22	21	20
PI	PO	PR	PW	SI	SO	SR	SW

Table 11. Fence mode encoding

fm **field**	**Mnemonic**	**Meaning**
0000	*none*	Normal Fence
1000	TSO	With `FENCE RW,RW`: exclude write-to-read ordering; otherwise: *Reserved for future use.*
other		*Reserved for future use.*

When the mode field *fm* is `0001` and both the predecessor and successor sets are 'RW', then the instruction acts as a special-case fence.tso. fence.tso orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the 'fence.tso's predecessor set unordered with non-AMO loads in its successor set.

When mode field *fm* is not `0001`, or when mode field *fm* is `0001` but the *pred* and *succ* fields are not both 'RW' (0x3), then the fence acts as a baseline fence (*e.g.*, *fm* is effectively `0000`). This is unaffected by the FIOM bits, described below (implicit promotion does not change how fence.tso is decoded).

The `xs1` and `xd` fields are unused and ignored.

In modes other than M-mode, fence is further affected by menvcfg.FIOM, senvcfg.FIOM<% if ext?(:H) %>, and/or henvcfg.FIOM<% end %> as follows:

Table 12. Effective PR/PW/SR/SW in (H)S-mode

menvcfg.FIOM	`pred.PI` → **effective** `PR`
	`pred.PO` → **effective** `PW`
	`succ.SI` → **effective** `SR`
	`succ.SO` → **effective** `SW`		

0	-		from encoding
1	0		from encoding
1	1		1

Table 13. Effective PR/PW/SR/SW in U-mode

menvcfg.FIOM	**senvcfg.FIOM**	`pred.PI` → **effective** `PR`
		`pred.PO` → **effective** `PW`
		`succ.SI` → **effective** `SR`
		`succ.SO` → **effective** `SW`		

0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding

menvcfg.FIOM	senvcfg.FIOM	pred.PI pred.PO succ.SI succ.SO	→ → → →	effective PR effective PW effective SR effective SW
1	-	1		1

<%- if ext?(:H) -%> .Effective PR/PW/SR/SW in VS-mode and VU-mode

menvcfg.FIOM	henvcfg.FIOM	pred.PI pred.PO succ.SI succ.SO	→ → → →	effective PR effective PW effective SR effective SW
0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding
1	-	1		1

<%- end -%>

B.127.3. Access

M
Always

B.127.4. Decode Variables

```
Bits<4> fm = $encoding[31:28];
Bits<4> pred = $encoding[27:24];
Bits<4> succ = $encoding[23:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.127.5. IDL Operation

```
Boolean is_pause;
if (implemented?(ExtensionName::Zihintpause)) {
  if ((pred == 1) && (succ == 0) && (xd == 0) && (xs1 == 0)) {
    is_pause = true;
  }
}
Boolean pred_i = pred[3] == 1;
Boolean pred_o = pred[2] == 1;
Boolean pred_r = pred[1] == 1;
Boolean pred_w = pred[0] == 1;
Boolean succ_i = succ[3] == 1;
Boolean succ_o = succ[2] == 1;
Boolean succ_r = succ[1] == 1;
Boolean succ_w = succ[0] == 1;
if (is_pause) {
  pause();
} else {
  if (mode() == PrivilegeMode::S) {
    if (menvcfg.FIOM == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
```

```
  } else if (mode() == PrivilegeMode::U) {
    if ((menvcfg.FIOM | senvcfg.FIOM) == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  } else if (mode() == PrivilegeMode::VS || mode() == PrivilegeMode::VU) {
    if ((menvcfg.FIOM | henvcfg.FIOM) == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  }
  fence(pred_i, pred_o, pred_r, pred_w, succ_i, succ_o, succ_r, succ_w);
}
```

B.127.6. Sail Operation

```
{
  // If the FIOM bit in menvcfg/senvcfg is set then the I/O bits can imply R/W.
  let fiom = is_fiom_active();
  let pred = effective_fence_set(pred, fiom);
  let succ = effective_fence_set(succ, fiom);

  match (pred, succ) {
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b11) => __barrier(Barrier_RISCV_rw_rw()),
    (_ : bits(2) @ 0b10, _ : bits(2) @ 0b11) => __barrier(Barrier_RISCV_r_rw()),
    (_ : bits(2) @ 0b10, _ : bits(2) @ 0b10) => __barrier(Barrier_RISCV_r_r()),
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b01) => __barrier(Barrier_RISCV_rw_w()),
    (_ : bits(2) @ 0b01, _ : bits(2) @ 0b01) => __barrier(Barrier_RISCV_w_w()),
    (_ : bits(2) @ 0b01, _ : bits(2) @ 0b11) => __barrier(Barrier_RISCV_w_rw()),
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b10) => __barrier(Barrier_RISCV_rw_r()),
    (_ : bits(2) @ 0b10, _ : bits(2) @ 0b01) => __barrier(Barrier_RISCV_r_w()),
    (_ : bits(2) @ 0b01, _ : bits(2) @ 0b10) => __barrier(Barrier_RISCV_w_r()),

    (_ : bits(4)       , _ : bits(2) @ 0b00) => (),
    (_ : bits(2) @ 0b00, _ : bits(4)       ) => (),

    _ => { print("FIXME: unsupported fence");
           () }
  };
  RETIRE_SUCCESS
}
```

B.127.7. Exceptions

This instruction does not generate synchronous exceptions.

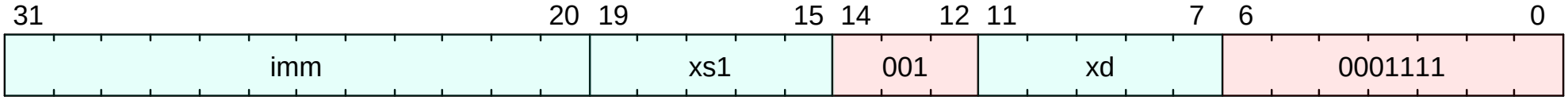
B.128. fence.i

Instruction fence

This instruction is defined by:

Zifencei, version >= Zifencei@2.0.0

B.128.1. Encoding



B.128.2. Description

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction. A FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does *not* ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart also has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I.

The unused fields in the FENCE.I instruction, *imm[11:0]*, *xs1*, and *xd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

Because FENCE.I only orders stores with a hart's own instruction fetches, application code should only rely upon FENCE.I if the application thread will not be migrated to a different hart. The EEI can provide mechanisms for efficient multiprocessor instruction-stream synchronization.

B.128.3. Access

M
Always

B.128.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.128.5. IDL Operation

```
ifence();
```

B.128.6. Sail Operation

```
{ /* __barrier(Barrier_RISCV_i); */ RETIRE_SUCCESS }
```

B.128.7. Exceptions

This instruction does not generate synchronous exceptions.

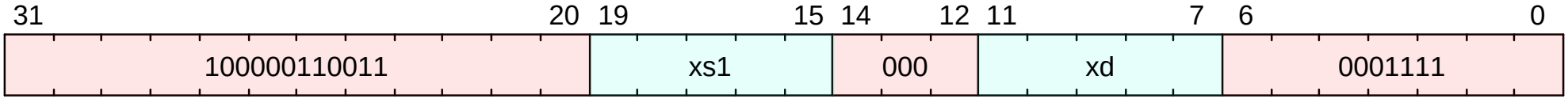
B.129. fence.tso

Memory ordering fence, total store ordering

This instruction is defined by:

I, version >= I@2.1.0

B.129.1. Encoding



B.129.2. Description

Orders memory operations.

fence.tso orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the 'fence.tso's predecessor set unordered with non-AMO loads in its successor set.

The `xs1` and `xd` fields are unused and ignored.

In modes other than M-mode, fence.tso is further affected by menvcfg.FIOM, senvcfg.FIOM<% if ext?(:H) %>, and/or henvcfg.FIOM<% end %>.

B.129.3. Access

M
Always

B.129.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.129.5. IDL Operation

```
fence_tso();
```

B.129.6. Sail Operation

```
{
  match (pred, succ) {
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b11) => sail_barrier(Barrier_RISCV_tso),
    (_ : bits(2) @ 0b00, _ : bits(2) @ 0b00) => (),

    _ => { print("FIXME: unsupported fence");
           () }
  };
  RETIRE_SUCCESS
}
```

B.129.7. Exceptions

This instruction does not generate synchronous exceptions.

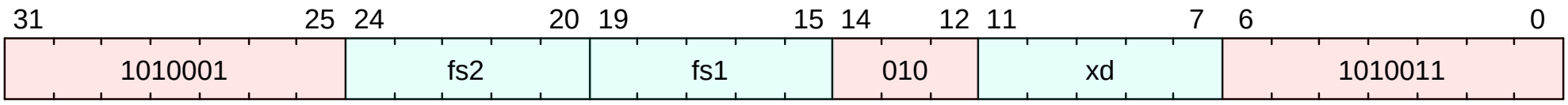
B.130. feq.d

Floating-Point Equal Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.130.1. Encoding



B.130.2. Description

The feq.d instruction writes 1 to `xd` if `fs1` and `fs2` are equal, and 0 otherwise. It is defined analogously to its single-precision counterpart, but operates on double-precision operands.

B.130.3. Access

M
Always

B.130.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.130.5. IDL Operation

```

```

B.130.6. Exceptions

This instruction does not generate synchronous exceptions.

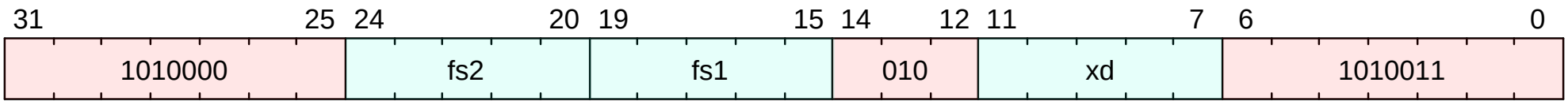
B.131. feq.s

Floating-Point Equal Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.131.1. Encoding



B.131.2. Description

The feq.s instruction writes 1 to `xd` if `fs1` and `fs2` are equal, and 0 otherwise. If either operand is NaN, the result is 0 (not equal). If either operand is a signaling NaN, the invalid flag is set. Positive zero is considered equal to negative zero.

B.131.3. Access

M
Always

B.131.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.131.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs1][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  if (is_sp_signaling_nan?(sp_value_a) || is_sp_signaling_nan?(sp_value_b)) {
    set_fp_flag(FpFlag::NV);
  }
  X[xd] = 0;
} else {
  X[xd] = sp_value_a == sp_value_b) || ((sp_value_a | sp_value_b)[30:0] == 0 ? 1 : 0;
}
```

B.131.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.131.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

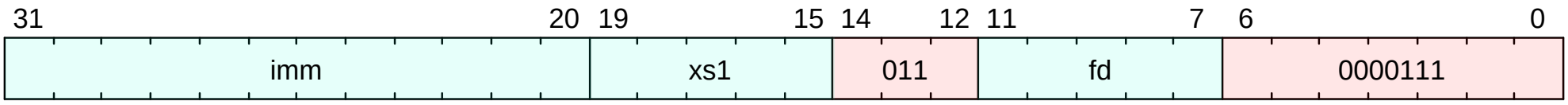
B.132. fld

Floating-Point Load Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.132.1. Encoding



B.132.2. Description

The fld instruction loads a double-precision floating-point value from memory into floating-point register `fd`. It is guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64. It doesn't modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

B.132.3. Access

M
Always

B.132.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.132.5. IDL Operation

```

```

B.132.6. Exceptions

This instruction does not generate synchronous exceptions.

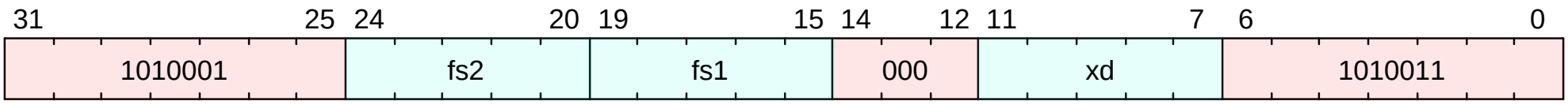
B.133. fle.d

Floating-Point Less Than or Equal Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.133.1. Encoding



B.133.2. Description

The fle.d instruction writes 1 to `xd` if `fs1` is less than or equal to `fs2`, and 0 otherwise. It is defined analogously to its single-precision counterpart, but operates on double-precision operands.

B.133.3. Access

M
Always

B.133.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.133.5. IDL Operation

```

```

B.133.6. Exceptions

This instruction does not generate synchronous exceptions.

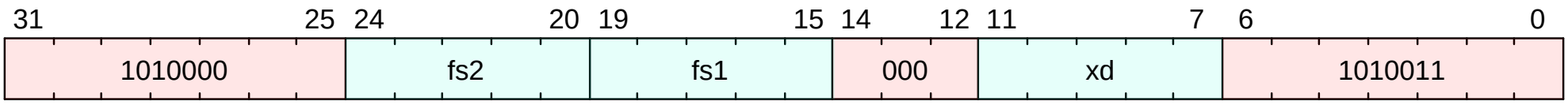
B.134. fle.s

Floating-Point Less Than or Equal Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.134.1. Encoding



B.134.2. Description

The fle.s instruction writes 1 to `xd` if `fs1` is less than or equal to `fs2`, and 0 otherwise. If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set. Positive zero and negative zero are considered equal.

B.134.3. Access

M
Always

B.134.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.134.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs2][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  if (is_sp_signaling_nan?(sp_value_a) || is_sp_signaling_nan?(sp_value_b)) {
    set_fp_flag(FpFlag::NV);
  }
  X[xd] = 0;
} else {
  X[xd] = sp_value_a == sp_value_b) || ((sp_value_a | sp_value_b)[30:0] == 0 ? 1 : 0;
}
```

B.134.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.134.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

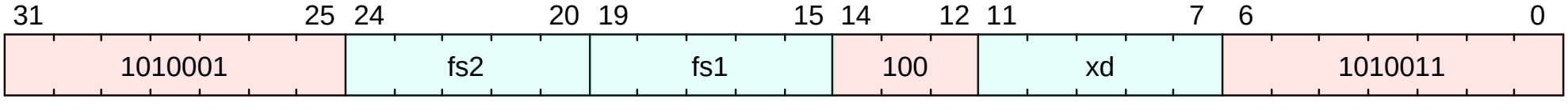
B.135. fleq.d

Floating-Point Less Than or Equal Quiet Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.135.1. Encoding



B.135.2. Description

The fleq.d instruction is defined like the fle.d instruction, except that quiet NaN inputs do not cause the invalid operation exception flag to be set.

B.135.3. Access

M
Always

B.135.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.135.5. IDL Operation

```

```

B.135.6. Exceptions

This instruction does not generate synchronous exceptions.

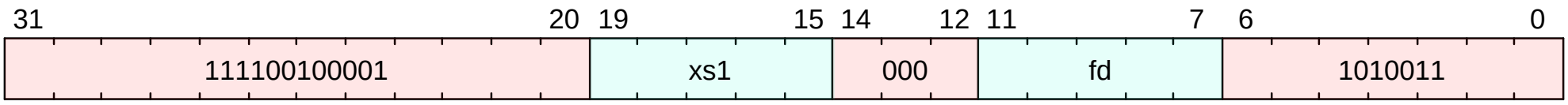
B.136. fli.d

Floating-Point Load Immediate Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.136.1. Encoding



B.136.2. Description

The fli.d instruction instruction loads one of 32 double-precision floating-point constants, encoded in the `xs1` field, into floating-point register `fd`. fli.d is encoded like fli.s, but with `fmt` set to D.

B.136.3. Access

M
Always

B.136.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.136.5. IDL Operation

```

```

B.136.6. Exceptions

This instruction does not generate synchronous exceptions.

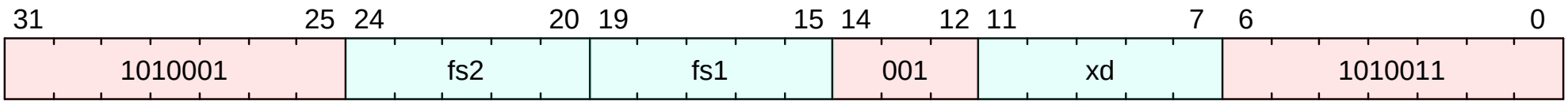
B.137. flt.d

Floating-Point Less Than Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.137.1. Encoding



B.137.2. Description

The flt.d instruction writes 1 to `xd` if `fs1` is less than `fs2`, and 0 otherwise. It is defined analogously to its single-precision counterpart, but operates on double-precision operands.

B.137.3. Access

M
Always

B.137.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.137.5. IDL Operation

```

```

B.137.6. Exceptions

This instruction does not generate synchronous exceptions.

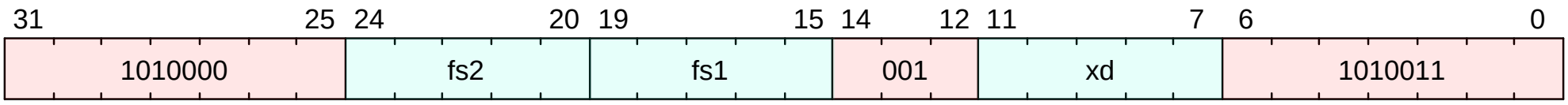
B.138. flt.s

Floating-Point Less Than Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.138.1. Encoding



B.138.2. Description

The flt.s instruction writes 1 to `xd` if `fs1` is less than `fs2`, and 0 otherwise. If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set.

B.138.3. Access

M
Always

B.138.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.138.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs2][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  set_fp_flag(FpFlag::NV);
  X[xd] = 0;
} else {
  Boolean sign_a = sp_value_a[31] == 1;
  Boolean sign_b = sp_value_b[31] == 1;
  Boolean a_lt_b = (sign_a != sign_b) ? (sign_a && sp_value_a[30:0] | sp_value_b[30:0]) != 0 : sp_value_a != sp_value_b) &&
(sign_a != (sp_value_a < sp_value_b);
  X[xd] = a_lt_b ? 1 : 0;
}
```

B.138.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.138.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

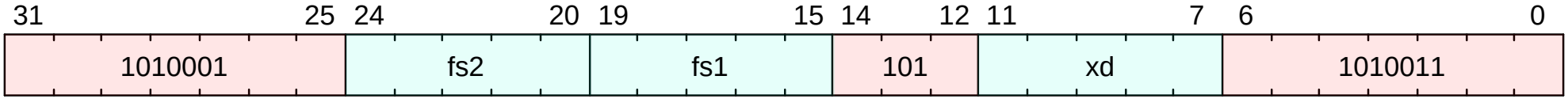
B.139. fltq.d

Floating-Point Less Than Quiet Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.139.1. Encoding



B.139.2. Description

The fltq.d instruction is defined like the flt.d instruction, except that quiet NaN inputs do not cause the invalid operation exception flag to be set.

B.139.3. Access

M
Always

B.139.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.139.5. IDL Operation

```

```

B.139.6. Exceptions

This instruction does not generate synchronous exceptions.

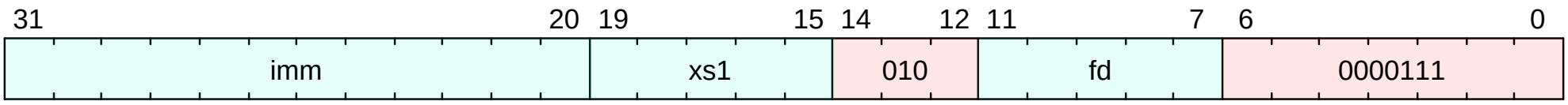
B.140. flw

Floating-Point Load Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.140.1. Encoding



B.140.2. Description

The flw instruction loads a single-precision floating-point value from memory at address `xs1` + `imm` into floating-point register `fd`. It does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

B.140.3. Access

M
Always

B.140.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.140.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[xs1] + $signed(imm);
Bits<32> sp_value = read_memory<32>(virtual_address, $encoding);
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.140.6. Sail Operation

```
{
let offset : xlenbits = sign_extend(imm);
/* Get the address, X(rs1) + offset.
   Some extensions perform additional checks on address validity. */
match ext_data_get_addr(rs1, offset, Read(Data), width) {
  Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
  Ext_DataAddr_OK(vaddr) =>
    if   check_misaligned(vaddr, width)
    then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
    else match translateAddr(vaddr, Read(Data)) {
      TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      TR_Address(addr, _) => {
        let (aq, rl, res) = (false, false, false);
        match (width) {
          BYTE => { handle_illegal(); RETIRE_FAIL },
          HALF =>
             process_fload16(rd, vaddr, mem_read(Read(Data), addr, 2, aq, rl, res)),
          WORD =>
             process_fload32(rd, vaddr, mem_read(Read(Data), addr, 4, aq, rl, res)),
          DOUBLE if sizeof(flen) >= 64 =>
             process_fload64(rd, vaddr, mem_read(Read(Data), addr, 8, aq, rl, res)),
          _ => report_invalid_width(__FILE__, __LINE__, width, "floating point load"),
        }
      }
    }
```

```
  }
}
```

B.140.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

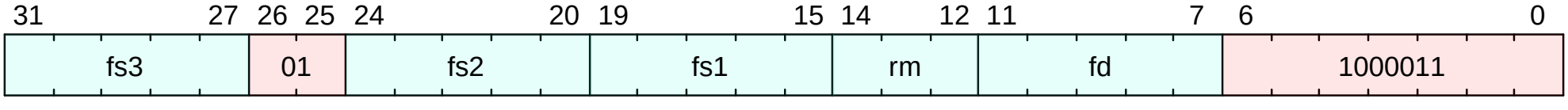
B.141. fmadd.d

Floating-Point Multiply-Add Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.141.1. Encoding



B.141.2. Description

The fmadd.d instruction multiplies the values in `fs1` and `fs2`, adds the value in `fs3`, and writes the final result to `fd`.

B.141.3. Access

M
Always

B.141.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.141.5. IDL Operation

```

```

B.141.6. Exceptions

This instruction does not generate synchronous exceptions.

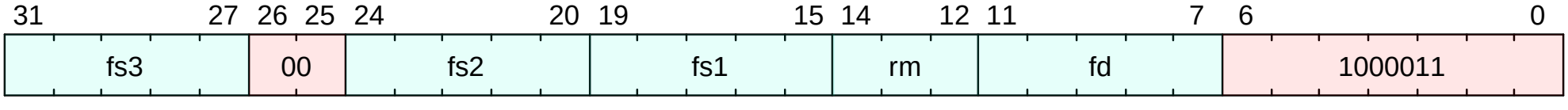
B.142. fmadd.s

Floating-Point Multiply-Add Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.142.1. Encoding



B.142.2. Description

The fmadd.s multiplies the values in `fs1` and `fs2`, adds the value in `fs3`, and writes the final result to `fd`.

B.142.3. Access

M
Always

B.142.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.142.5. IDL Operation

```

```

B.142.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.142.7. Exceptions

This instruction does not generate synchronous exceptions.

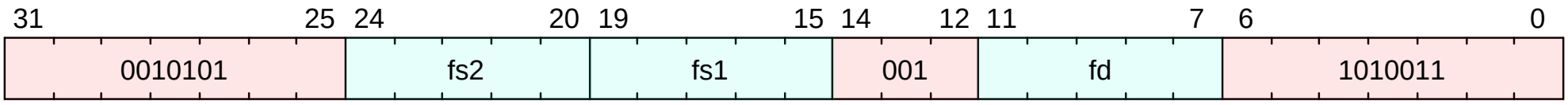
B.143. fmax.d

Floating-Point Maximum-Number Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.143.1. Encoding



B.143.2. Description

The fmax.d instruction writes larger of `fs1` and `fs2` to `fd`. For the purposes of this instruction, the value `-0.0` is considered to be less than the value `+0.0`. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

B.143.3. Access

M
Always

B.143.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.143.5. IDL Operation

```

```

B.143.6. Exceptions

This instruction does not generate synchronous exceptions.

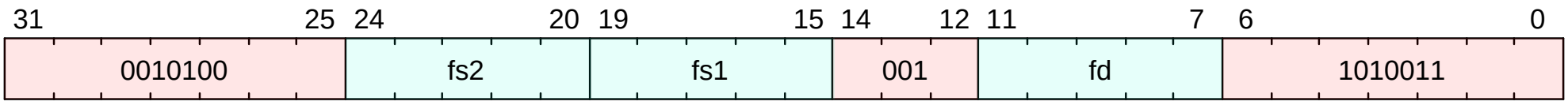
B.144. fmax.s

Floating-Point Maximum-Number Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.144.1. Encoding



B.144.2. Description

The fmax.s instruction writes larger of `fs1` and `fs2` to `fd`. For the purposes of this instruction, the value `-0.0` is considered to be less than the value `+0.0`. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

Note that in version 2.2 of the F extension, the fmin.s and fmax.s instructions were amended to implement the proposed *IEEE 754-201x* `minimumNumber` and `maximumNumber` operations, rather than the *IEEE 754-2008 minNum* and *maxNum* operations. These operations differ in their handling of signaling NaNs.

B.144.3. Access

M
Always

B.144.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.144.5. IDL Operation

```

```

B.144.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.144.7. Exceptions

This instruction does not generate synchronous exceptions.

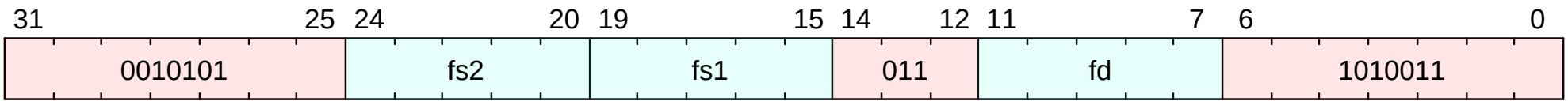
B.145. fmaxm.d

Floating-Point Maximum-Number NaN Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.145.1. Encoding



B.145.2. Description

The fmaxm.d instruction writes larger of `fs1` and `fs2` to `fd`. It is defined like fmax.d instruction, except that if either input is NaN, the result is the canonical NaN.

B.145.3. Access

M
Always

B.145.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.145.5. IDL Operation

```

```

B.145.6. Exceptions

This instruction does not generate synchronous exceptions.

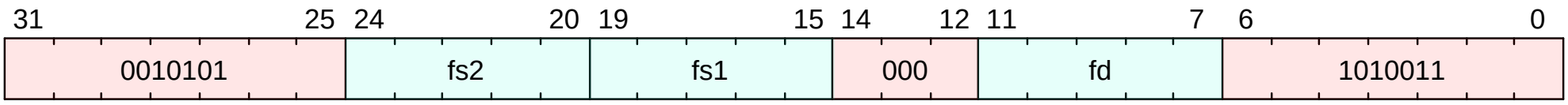
B.146. fmin.d

Floating-Point Minimum-Number Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.146.1. Encoding



B.146.2. Description

The fmin.d instruction writes smaller of `fs1` and `fs2` to `fd`. For the purposes of this instruction, the value `-0.0` is considered to be less than the value `+0.0`. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

B.146.3. Access

M
Always

B.146.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.146.5. IDL Operation

```

```

B.146.6. Exceptions

This instruction does not generate synchronous exceptions.

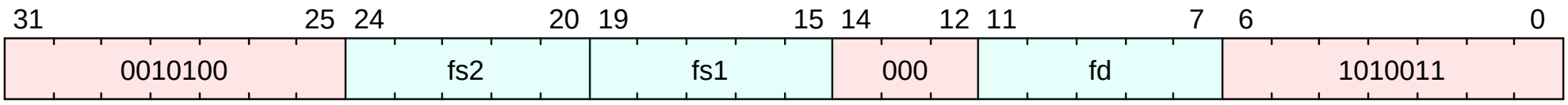
B.147. fmin.s

Floating-Point Minimum-Number Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.147.1. Encoding



B.147.2. Description

The fmin.s instruction writes smaller of `fs1` and `fs2` to `fd`. For the purposes of this instruction, the value `-0.0` is considered to be less than the value `+0.0`. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

Note that in version 2.2 of the F extension, the fmin.s and fmax.s instructions were amended to implement the proposed *IEEE 754-201x* `minimumNumber` and `maximumNumber` operations, rather than the *IEEE 754-2008 minNum* and *maxNum* operations. These operations differ in their handling of signaling NaNs.

B.147.3. Access

M
Always

B.147.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.147.5. IDL Operation

```

```

B.147.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.147.7. Exceptions

This instruction does not generate synchronous exceptions.

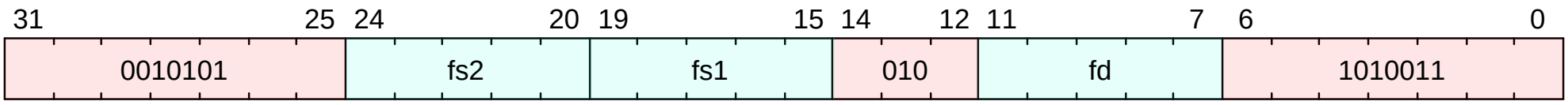
B.148. fminm.d

Floating-Point Minimum-Number NaN Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.148.1. Encoding



B.148.2. Description

The fminm.d instruction writes smaller of `fs1` and `fs2` to `fd`. It is defined like fmin.d instruction, except that if either input is NaN, the result is the canonical NaN.

B.148.3. Access

M
Always

B.148.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.148.5. IDL Operation

```

```

B.148.6. Exceptions

This instruction does not generate synchronous exceptions.

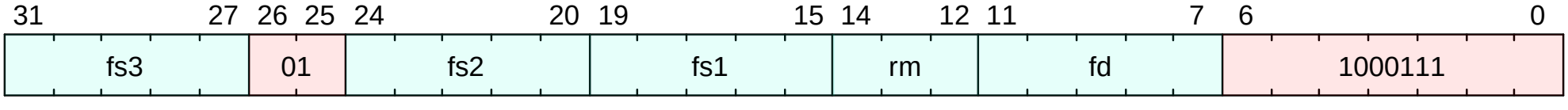
B.149. fmsub.d

Floating-Point Multiply-Subtract Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.149.1. Encoding



B.149.2. Description

The fmsub.d instruction multiplies the values in `fs1` and `fs2`, subtracts the value in `fs3`, and writes the final result to `fd`.

B.149.3. Access

M
Always

B.149.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.149.5. IDL Operation

```

```

B.149.6. Exceptions

This instruction does not generate synchronous exceptions.

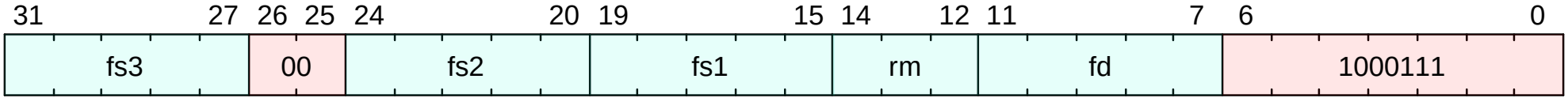
B.150. fmsub.s

Floating-Point Multiply-Subtract Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.150.1. Encoding



B.150.2. Description

The fmsub.s multiplies the values in `fs1` and `fs2`, subtracts the value in `fs3`, and writes the final result to `fd`.

B.150.3. Access

M
Always

B.150.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.150.5. IDL Operation

```

```

B.150.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.150.7. Exceptions

This instruction does not generate synchronous exceptions.

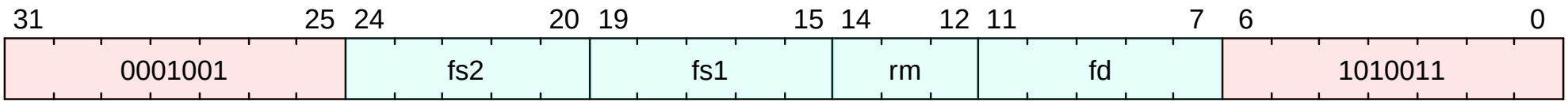
B.151. fmul.d

Floating-Point Multiply Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.151.1. Encoding



B.151.2. Description

The fmul.d instruction performs the double-precision floating-point multiplication between `fs1` and `fs2`. It is defined analogously to its single-precision counterpart, but operates on double-precision operands and produces double-precision results.

B.151.3. Access

M
Always

B.151.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.151.5. IDL Operation

```

```

B.151.6. Exceptions

This instruction does not generate synchronous exceptions.

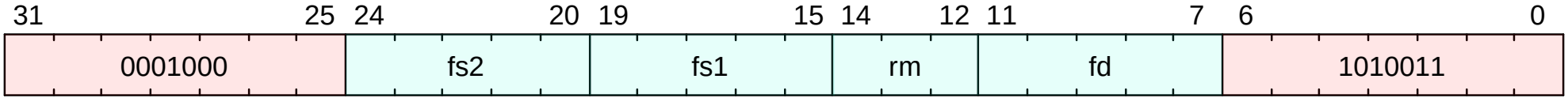
B.152. fmul.s

Floating-Point Multiply Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.152.1. Encoding



B.152.2. Description

The fmul.s instruction performs the single-precision floating-point multiplication between `fs1` and `fs2`, and writes the result in `fd`.

B.152.3. Access

M
Always

B.152.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.152.5. IDL Operation

```

```

B.152.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) = match op {
        FADD_S  => riscv_f32Add (rm_3b, rs1_val_32b, rs2_val_32b),
        FSUB_S  => riscv_f32Sub (rm_3b, rs1_val_32b, rs2_val_32b),
        FMUL_S  => riscv_f32Mul (rm_3b, rs1_val_32b, rs2_val_32b),
        FDIV_S  => riscv_f32Div (rm_3b, rs1_val_32b, rs2_val_32b)
      };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.152.7. Exceptions

This instruction does not generate synchronous exceptions.

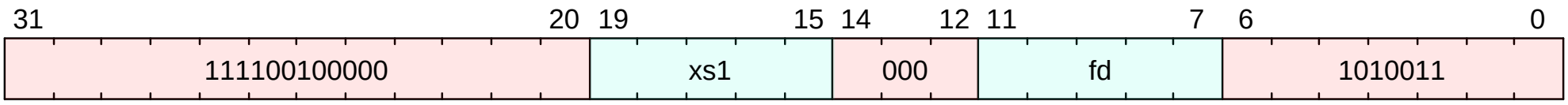
B.153. fmv.d.x

Floating-Point Move from Integer Register to Double-Precision Register

This instruction is defined by:

D, version >= D@2.2.0

B.153.1. Encoding



B.153.2. Description

The fmv.d.x instruction moves the double-precision value encoded in `IEEE 754-2008` standard encoding from the integer register `xs1` to the floating-point register `fd`.

B.153.3. Access

M
Always

B.153.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.153.5. IDL Operation

```

```

B.153.6. Exceptions

This instruction does not generate synchronous exceptions.

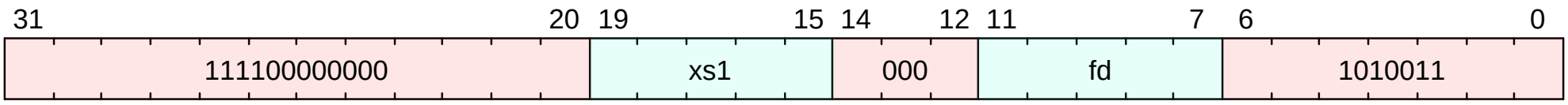
B.154. fmv.w.x

Floating-Point Move Single-Precision Word from Integer Register

This instruction is defined by:

F, version >= F@2.2.0

B.154.1. Encoding



B.154.2. Description

The fmv.w.x instruction moves the single-precision value encoded in `IEEE 754-2008` standard encoding from the lower 32 bits of integer register `xs1` to the floating-point register `fd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

B.154.3. Access

M
Always

B.154.4. Decode Variables

```
Bits<5> xs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.154.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = X[xs1][31:0];
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.154.6. Sail Operation

```
{
  let rs1_val_X            = X(rs1);
  let rd_val_S             = rs1_val_X [31..0];
  F(rd) = nan_box (rd_val_S);
  RETIRE_SUCCESS
}
```

B.154.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

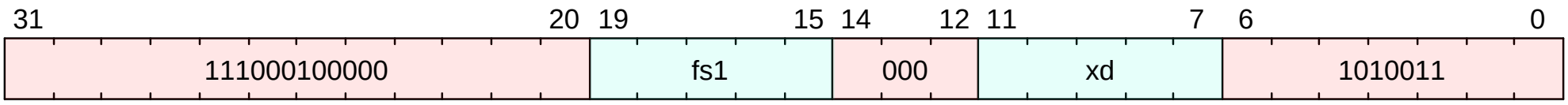
B.155. fmv.x.d

Floating-Point Move from Double-Precision Register to Integer Register

This instruction is defined by:

D, version >= D@2.2.0

B.155.1. Encoding



B.155.2. Description

The fmv.x.d instruction moves the double-precision value in floating-point register `fs1` to a representation in `IEEE 754-2008` standard encoding in integer register `xd`.

B.155.3. Access

M
Always

B.155.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.155.5. IDL Operation

```

```

B.155.6. Exceptions

This instruction does not generate synchronous exceptions.

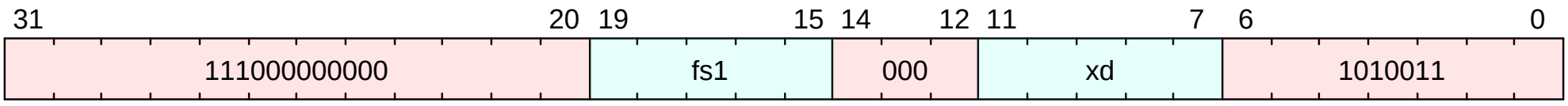
B.156. fmv.x.w

Floating-Point Move Single-Precision Word to Integer Register

This instruction is defined by:

F, version >= F@2.2.0

B.156.1. Encoding



B.156.2. Description

The fmv.x.w instruction moves the single-precision value in floating-point register `fs1` represented in `IEEE 754-2008` encoding to the lower 32 bits of integer register `xd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.

B.156.3. Access

M
Always

B.156.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.156.5. IDL Operation

```
check_f_ok($encoding);
X[xd] = sext(f[fs1][31:0], 32);
```

B.156.6. Sail Operation

```
{
  let rs1_val_X            = X(rs1);
  let rd_val_S             = rs1_val_X [31..0];
  F(rd) = nan_box (rd_val_S);
  RETIRE_SUCCESS
}
```

B.156.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

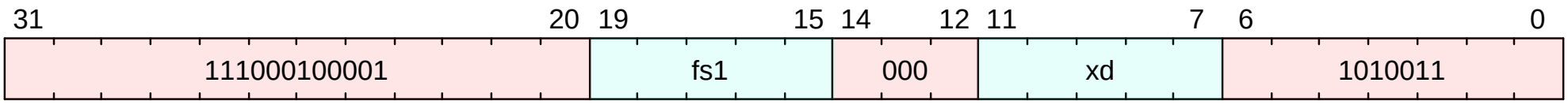
B.157. fmvh.x.d

Floating-Point Move High Half from Double-Precision Register to Integer Register

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.157.1. Encoding



B.157.2. Description

The fmvh.x.d instruction moves bits 63:32 of floating-point register `fs1` into integer register `xd`. fmvh.x.d is used in conjunction with the existing fmv.x.w instruction to move a double-precision floating-point number to a pair of integer-registers.

B.157.3. Access

M
Always

B.157.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.157.5. IDL Operation

```

```

B.157.6. Exceptions

This instruction does not generate synchronous exceptions.

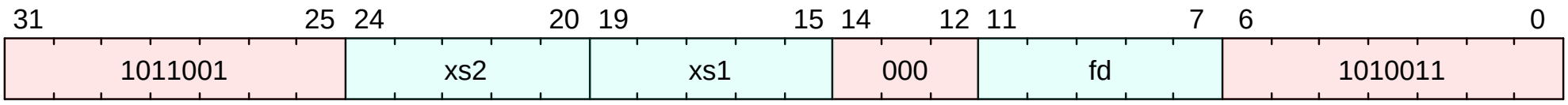
B.158. fmvp.d.x

Floating-Point Move Pair from Integer Registers to Double-Precision Register

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.158.1. Encoding



B.158.2. Description

The fmvp.d.x instruction moves a double-precision number from a pair of integer registers into a floating-point register. Integer registers `xs1` and `xs2` supply bits 31:0 and 63:32, respectively; the result is written to floating-point register `fd`.

B.158.3. Access

M
Always

B.158.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.158.5. IDL Operation

```

```

B.158.6. Exceptions

This instruction does not generate synchronous exceptions.

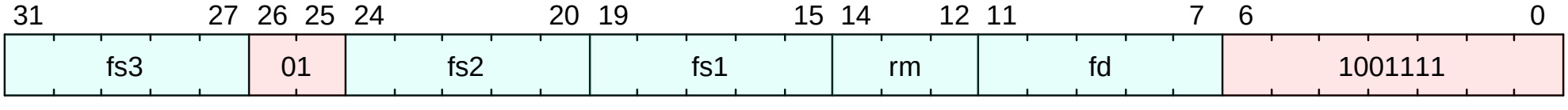
B.159. fnmadd.d

Floating-Point Negate-Multiply-Add Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.159.1. Encoding



B.159.2. Description

The fnmadd.d instruction multiplies the values in `fs1` and `fs2`, negates the product, subtracts the value in `fs3`, and writes the final result to `fd`.

B.159.3. Access

M
Always

B.159.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.159.5. IDL Operation

```

```

B.159.6. Exceptions

This instruction does not generate synchronous exceptions.

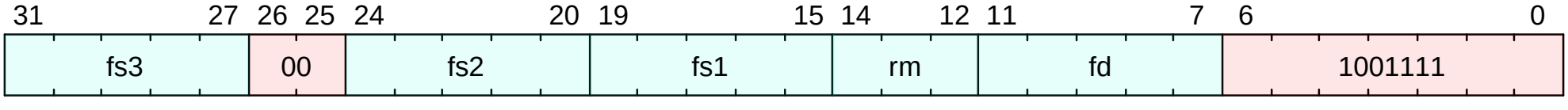
B.160. fnmadd.s

Floating-Point Negate-Multiply-Add Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.160.1. Encoding



B.160.2. Description

The fnmadd.s multiplies the values in `fs1` and `fs2`, negates the product, subtracts the value in `fs3`, and writes the final result to `fd`.

B.160.3. Access

M
Always

B.160.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.160.5. IDL Operation

```

```

B.160.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.160.7. Exceptions

This instruction does not generate synchronous exceptions.

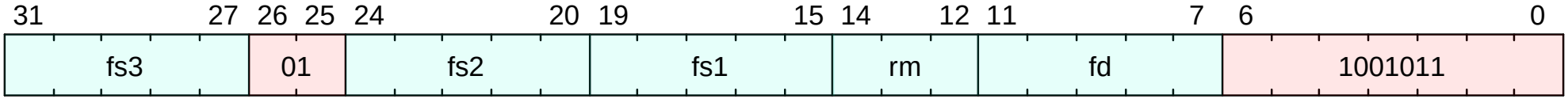
B.161. fnmsub.d

Floating-Point Negate-Multiply-Subtract Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.161.1. Encoding



B.161.2. Description

The fnmsub.d instruction multiplies the values in `fs1` and `fs2`, negates the product, adds the value in `fs3`, and writes the final result to `fd`.

B.161.3. Access

M
Always

B.161.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.161.5. IDL Operation

```

```

B.161.6. Exceptions

This instruction does not generate synchronous exceptions.

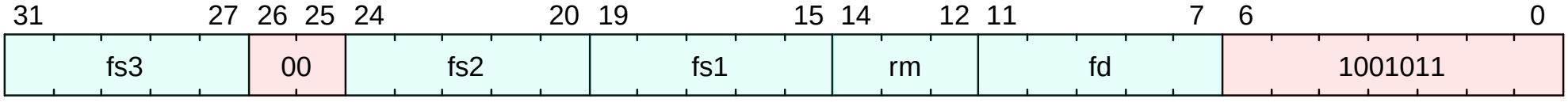
B.162. fnmsub.s

Floating-Point Negate-Multiply-Subtract Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.162.1. Encoding



B.162.2. Description

The fnmsub.s instruction multiplies the values in `fs1` and `fs2`, negates the product, adds the value in `fs3`, and writes the final result to `fd`.

B.162.3. Access

M
Always

B.162.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.162.5. IDL Operation

```

```

B.162.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.162.7. Exceptions

This instruction does not generate synchronous exceptions.

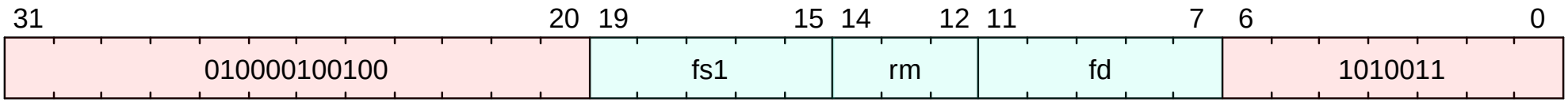
B.163. fround.d

Floating-Point Round Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.163.1. Encoding



B.163.2. Description

The fround.d instruction rounds the double-precision floating-point number in floating-point register `fs1` to an integer, according to the rounding mode specified in the instruction's `rm` field. It then writes that integer, represented as a double-precision floating-point number, to floating-point register `fd`. Zero and infinite inputs are copied to `fd` unmodified. Signaling NaN inputs cause the invalid operation exception flag to be set; no other exception flags are set. fround.d is encoded like fcvt.d.s, but with `xs2` set to 4.

B.163.3. Access

M
Always

B.163.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.163.5. IDL Operation

```

```

B.163.6. Exceptions

This instruction does not generate synchronous exceptions.

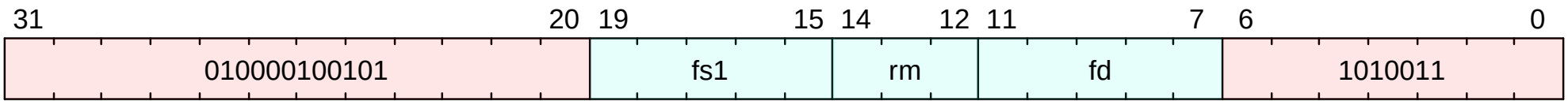
B.164. froundnx.d

Floating-Point Round-to-Integer Inexact Double-Precision

This instruction is defined by:

allOf: * D, version >= D@2.2.0 * Zfa, version >= Zfa@1.0.0

B.164.1. Encoding



B.164.2. Description

The froundnx.d instruction is defined in the same way as fround.d, but it also sets the inexact exception flag if the input differs from the rounded result and is not NaN. froundnx.d is encoded like fcvt.d.s, but with `xs2` set to 5.

B.164.3. Access

M
Always

B.164.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.164.5. IDL Operation

```

```

B.164.6. Exceptions

This instruction does not generate synchronous exceptions.

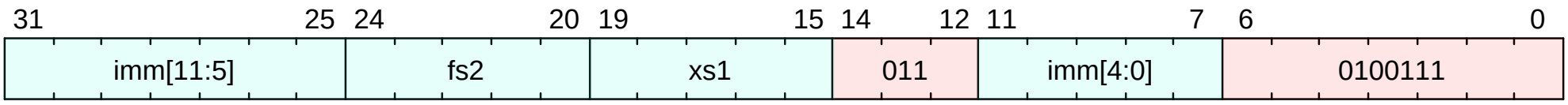
B.165. fsd

Floating-Point Store Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.165.1. Encoding



B.165.2. Description

The fsd instruction stores a double-precision value from the floating-point registers to memory. It is guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64. It doesn't modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

B.165.3. Access

M
Always

B.165.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.165.5. IDL Operation

```

```

B.165.6. Exceptions

This instruction does not generate synchronous exceptions.

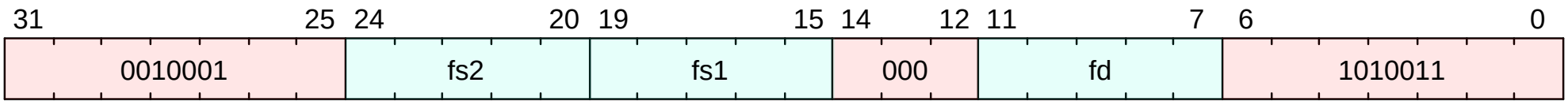
B.166. fsgnj.d

Floating-Point Sign-Inject Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.166.1. Encoding



B.166.2. Description

The fsgnj.d instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is taken from `fs2`'s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.166.3. Access

M
Always

B.166.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.166.5. IDL Operation

```

```

B.166.6. Exceptions

This instruction does not generate synchronous exceptions.

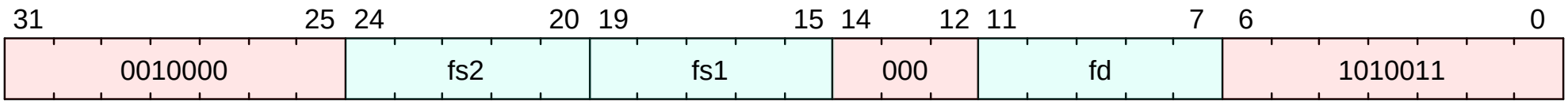
B.167. fsgnj.s

Floating-Point Sign-Inject Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.167.1. Encoding



B.167.2. Description

The fsgnj.s instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is taken from `fs2`'s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.167.3. Access

M
Always

B.167.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.167.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = {f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.167.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.167.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

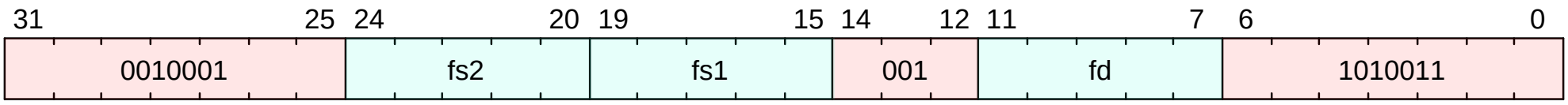
B.168. fsgnjn.d

Floating-Point Sign-Inject Negate Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.168.1. Encoding



B.168.2. Description

The fsgnjn.d instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is opposite of `fs2`'s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.168.3. Access

M
Always

B.168.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.168.5. IDL Operation

```

```

B.168.6. Exceptions

This instruction does not generate synchronous exceptions.

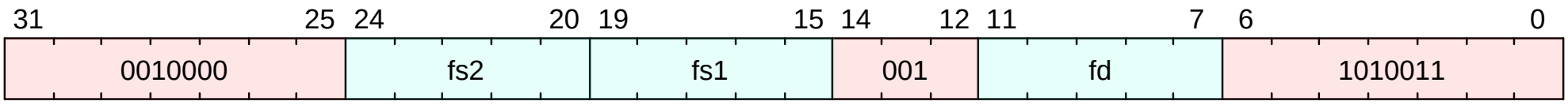
B.169. fsgnjn.s

Floating-Point Sign-Inject Negate Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.169.1. Encoding



B.169.2. Description

The fsgnjn.s instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is opposite of `fs2`'s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.169.3. Access

M
Always

B.169.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.169.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = {~f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.169.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.169.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

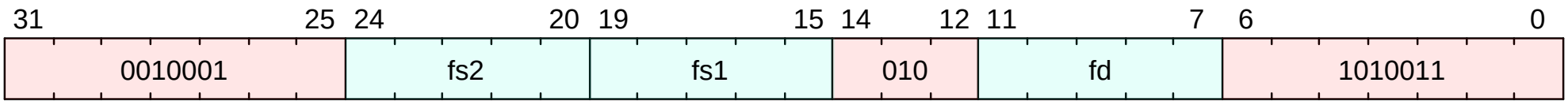
B.170. fsgnjx.d

Floating-Point Sign-Inject XOR Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.170.1. Encoding



B.170.2. Description

The fsgnjx.d instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is the XOR of sign bits of `fs1` and `fs2`, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.170.3. Access

M
Always

B.170.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.170.5. IDL Operation

```

```

B.170.6. Exceptions

This instruction does not generate synchronous exceptions.

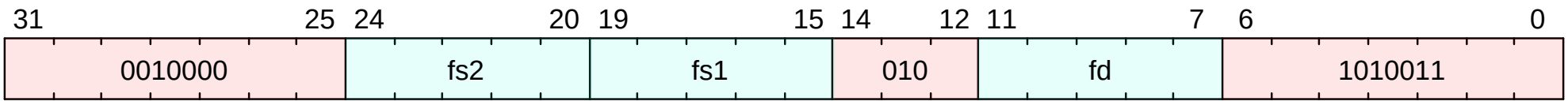
B.171. fsgnjx.s

Floating-Point Sign-Inject XOR Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.171.1. Encoding



B.171.2. Description

The fsgnjx.s instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is the XOR of sign bits of `fs1` and `fs2`, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

B.171.3. Access

M
Always

B.171.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> fd = $encoding[11:7];
```

B.171.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = {f[fs1][31] ^ f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

B.171.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

B.171.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

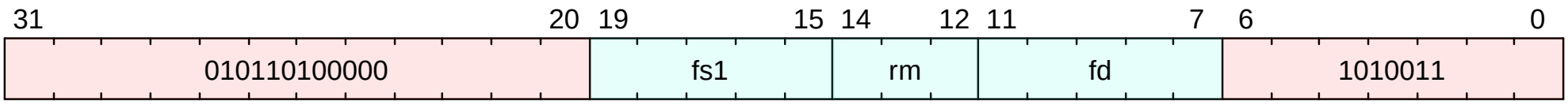
B.172. fsqrt.d

Floating-Point Square Root Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.172.1. Encoding



B.172.2. Description

The fsqrt.d instruction computes the square root of `fs1` and result is written in `fd`. It is defined analogously to its single-precision counterpart, but operates on double-precision operands and produces double-precision results.

B.172.3. Access

M
Always

B.172.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.172.5. IDL Operation

```

```

B.172.6. Exceptions

This instruction does not generate synchronous exceptions.

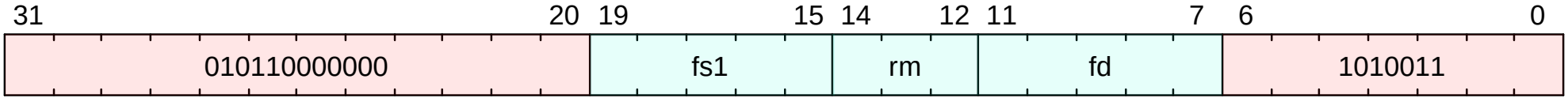
B.173. fsqrt.s

Floating-Point Square Root Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.173.1. Encoding



B.173.2. Description

The fsqrt.s instruction computes the square root of `fs1` and writes the result is written to `fd`.

B.173.3. Access

M
Always

B.173.4. Decode Variables

```
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.173.5. IDL Operation

```

```

B.173.6. Sail Operation

```
{
  assert(sizeof(xlen) >= 64);
  let rs1_val_LU = X(rs1)[63..0];
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_S) = riscv_ui64ToF32 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_S;
      RETIRE_SUCCESS
    }
  }
}
```

B.173.7. Exceptions

This instruction does not generate synchronous exceptions.

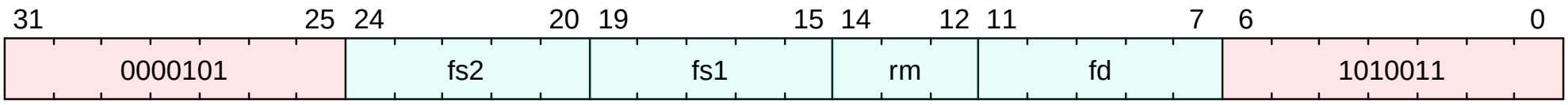
B.174. fsub.d

Floating-Point Subtract Double-Precision

This instruction is defined by:

D, version >= D@2.2.0

B.174.1. Encoding



B.174.2. Description

The fsub.d instruction is analogous to fsub.s and performs double-precision floating-point subtraction between `fs1` and `fs2` and writes the final result to `fd`.

B.174.3. Access

M
Always

B.174.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.174.5. IDL Operation

```

```

B.174.6. Exceptions

This instruction does not generate synchronous exceptions.

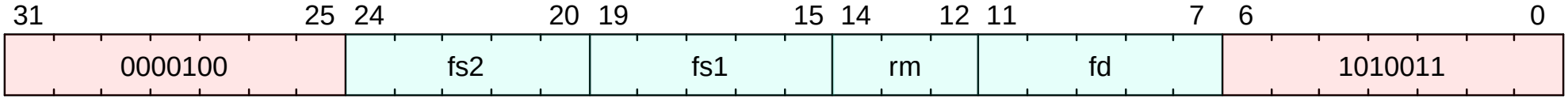
B.175. fsub.s

Floating-Point Subtract Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.175.1. Encoding



B.175.2. Description

The fsub.s instruction performs the single-precision floating-point subtraction of `fs2` from `fs1` and writes the result in `fd`.

B.175.3. Access

M
Always

B.175.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

B.175.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode mode = rm_to_mode(rm, $encoding);
f[fd] = f32_sub(f[fs1], f[fs2], mode);
```

B.175.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) = match op {
        FADD_S  => riscv_f32Add (rm_3b, rs1_val_32b, rs2_val_32b),
        FSUB_S  => riscv_f32Sub (rm_3b, rs1_val_32b, rs2_val_32b),
        FMUL_S  => riscv_f32Mul (rm_3b, rs1_val_32b, rs2_val_32b),
        FDIV_S  => riscv_f32Div (rm_3b, rs1_val_32b, rs2_val_32b)
      };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

B.175.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

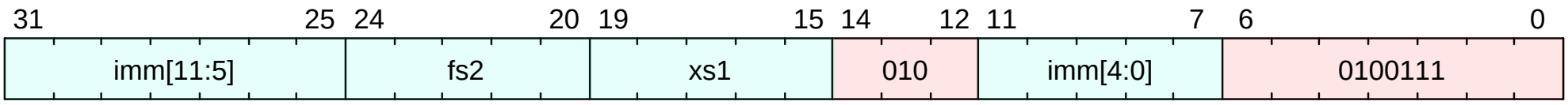
B.176. fsw

Floating-Point Store Single-Precision

This instruction is defined by:

F, version >= F@2.2.0

B.176.1. Encoding



B.176.2. Description

The fsw instruction stores a single-precision floating-point value in `fs2` to memory at address `xs1` + `imm`. It does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

B.176.3. Access

M
Always

B.176.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.176.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<32>(virtual_address, f[fs2][31:0], $encoding);
```

B.176.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  let (aq, rl, con) = (false, false, false);
  /* Get the address, X(rs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(rs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
    if   check_misaligned(vaddr, width)
    then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
    else match translateAddr(vaddr, Write(Data)) {
      TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      TR_Address(addr, _) => {
        let eares : MemoryOpResult(unit) = match width {
          BYTE   => MemValue () /* bogus placeholder for illegal size */,
          HALF   => mem_write_ea(addr, 2, aq, rl, false),
          WORD   => mem_write_ea(addr, 4, aq, rl, false),
          DOUBLE => mem_write_ea(addr, 8, aq, rl, false)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let rs2_val = F(rs2);
            match (width) {
              BYTE => { handle_illegal(); RETIRE_FAIL },
              HALF => process_fstore (vaddr, mem_write_value(addr, 2, rs2_val[15..0], aq, rl, con)),
              WORD => process_fstore (vaddr, mem_write_value(addr, 4, rs2_val[31..0], aq, rl, con)),
              DOUBLE if sizeof(flen) >= 64 =>
                process_fstore (vaddr, mem_write_value(addr, 8, rs2_val, aq, rl, con)),
              _ => report_invalid_width(__FILE__, __LINE__, width, "floating point store"),
            };
```

```
            }
          }
        }
      }
  }
}
```

B.176.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

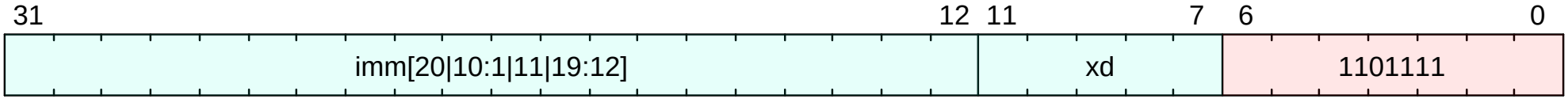
B.177. jal

Jump and link

This instruction is defined by:

I, version >= I@2.1.0

B.177.1. Encoding



B.177.2. Description

Jump to a PC-relative offset and store the return address in xd.

B.177.3. Access

M
Always

B.177.4. Decode Variables

```
signed Bits<21> imm = sext({$encoding[31], $encoding[19:12], $encoding[20], $encoding[30:21], 1'd0});
Bits<5> xd = $encoding[11:7];
```

B.177.5. IDL Operation

```
XReg return_addr = $pc + 4;
X[xd] = return_addr;
jump_halfword($pc + $signed(imm));
```

B.177.6. Sail Operation

```
{
  let t : xlenbits = PC + sign_extend(imm);
  /* Extensions get the first checks on the prospective target address. */
  match ext_control_check_pc(t) {
    Ext_ControlAddr_Error(e) => {
      ext_handle_control_check_error(e);
      RETIRE_FAIL
    },
    Ext_ControlAddr_OK(target) => {
      /* Perform standaxd alignment check */
      if bit_to_bool(target[1]) & not(extension("C"))
      then {
        handle_mem_exception(target, E_Fetch_Addr_Align());
        RETIRE_FAIL
      } else {
        X(xd) = get_next_pc();
        set_next_pc(target);
        RETIRE_SUCCESS
      }
    }
  }
}
```

B.177.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

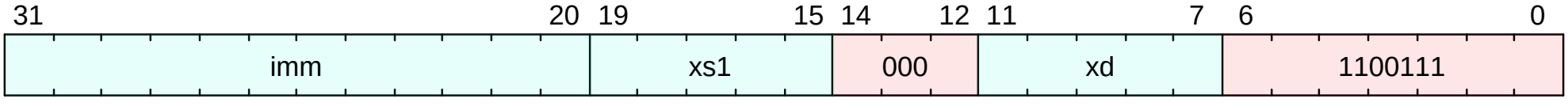
B.178. jalr

Jump and link register

This instruction is defined by:

I, version >= I@2.1.0

B.178.1. Encoding



B.178.2. Description

Jump to an address formed by adding xs1 to a signed offset then clearing the least significant bit, and store the return address in xd.

B.178.3. Access

M
Always

B.178.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.178.5. IDL Operation

```
XReg addr = (X[xs1] + $signed(imm)) & ~MXLEN'1;
XReg returnaddr;
returnaddr = $pc + 4;
X[xd] = returnaddr;
jump(addr);
```

B.178.6. Sail Operation

```
{
/* For the sequential model, the memory-model definition doesn't work directly
 * if xs1 = xd.  We would effectively have to keep a regfile for reads and another for
 * writes, and swap on instruction completion.  This could perhaps be optimized in
 * some manner, but for now, we just keep a reoxdered definition to improve simulator
 * performance.
 */
  let t : xlenbits = X(xs1) + sign_extend(imm);
  /* Extensions get the first checks on the prospective target address. */
  match ext_control_check_addr(t) {
    Ext_ControlAddr_Error(e) => {
      ext_handle_control_check_error(e);
      RETIRE_FAIL
    },
    Ext_ControlAddr_OK(addr) => {
      let target = [addr with 0 = bitzero];  /* clear addr[0] */
      if bit_to_bool(target[1]) & not(extension("C")) then {
        handle_mem_exception(target, E_Fetch_Addr_Align());
        RETIRE_FAIL
      } else {
        X(xd) = get_next_pc();
        set_next_pc(target);
        RETIRE_SUCCESS
      }
    }
  }
}
```

B.178.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

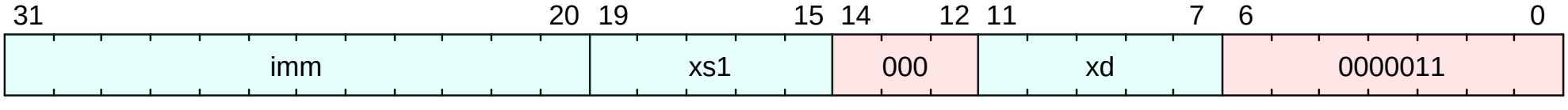
B.179. lb

Load byte

This instruction is defined by:

I, version >= I@2.1.0

B.179.1. Encoding



B.179.2. Description

Load 8 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Sign extend the result.

B.179.3. Access

M
Always

B.179.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.179.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = sext(read_memory<8>(virtual_address, $encoding), 8);
```

B.179.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.179.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

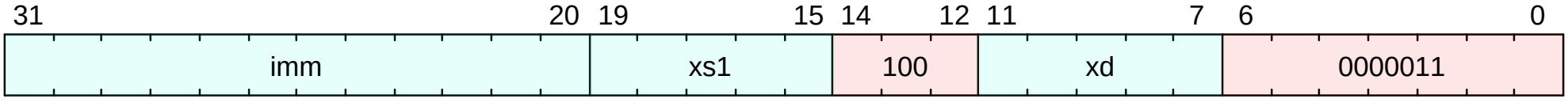
B.180. lbu

Load byte unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.180.1. Encoding



B.180.2. Description

Load 8 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Zero extend the result.

B.180.3. Access

M
Always

B.180.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.180.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = read_memory<8>(virtual_address, $encoding);
```

B.180.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.180.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

B.181. ld

Load doubleword

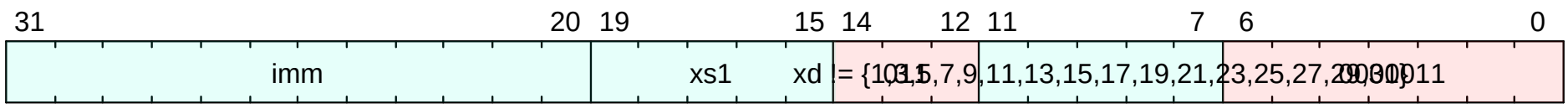
This instruction is defined by:

anyOf: * I, version >= I@2.1.0 * Zilsd, version >= Zilsd@1.0.0

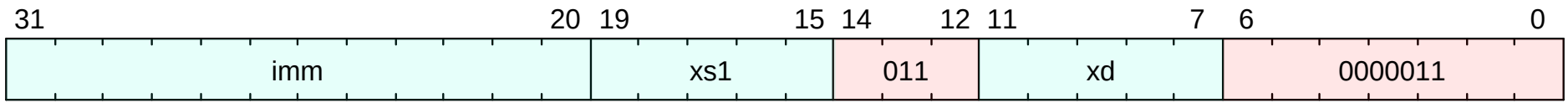
B.181.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.181.2. Description

For RV64, load 64 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. <% if ext?(:Zilsd) %> For RV32, Loads a 64-bit value into registers xd and xd+1. The effective address is obtained by adding register xs1 to the sign-extended 12-bit offset. <% end %>

B.181.3. Access

M
Always

B.181.4. Decode Variables

RV32

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

RV64

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.181.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
if (xlen() == 32) {
  if (implemented?(ExtensionName::Zilsd)) {
    Bits<64> data = read_memory<64>(virtual_address, $encoding);
    X[xd] = data[31:0];
    X[xd + 1] = data[63:32];
  } else {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else {
  X[xd] = read_memory<64>(virtual_address, $encoding);
}
```

B.181.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
```

```
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.181.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

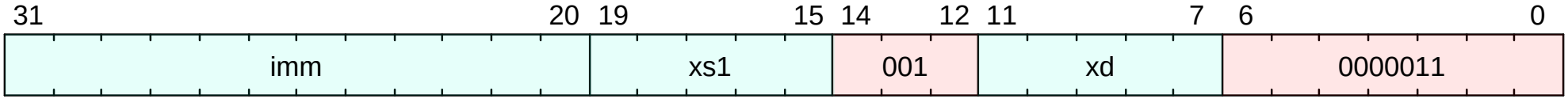
B.182. lh

Load halfword

This instruction is defined by:

I, version >= I@2.1.0

B.182.1. Encoding



B.182.2. Description

Load 16 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Sign extend the result.

B.182.3. Access

M
Always

B.182.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.182.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = sext(read_memory<16>(virtual_address, $encoding), 16);
```

B.182.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.182.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

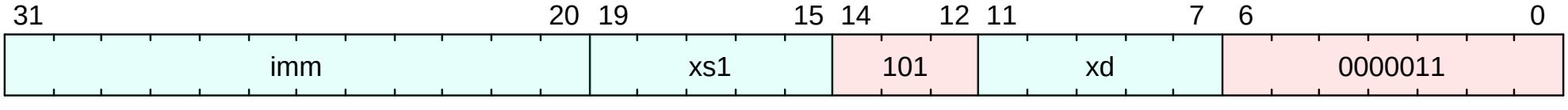
B.183. lhu

Load halfword unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.183.1. Encoding



B.183.2. Description

Load 16 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Zero extend the result.

B.183.3. Access

M
Always

B.183.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.183.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = read_memory<16>(virtual_address, $encoding);
```

B.183.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.183.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

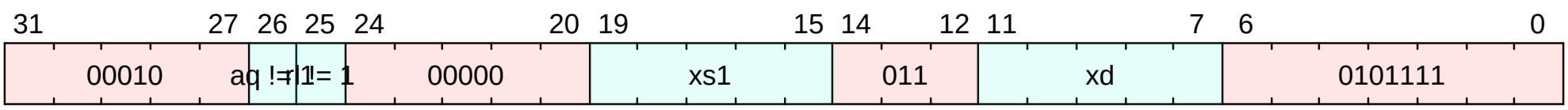
B.184. lr.d

Load reserved doubleword

This instruction is defined by:

Zalrsc, version >= Zalrsc@1.0.0

B.184.1. Encoding



B.184.2. Description

Loads a word from the address in xs1, places the value in xd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

The address in xs1 must be 8-byte aligned.

If the address is not naturally aligned, a `LoadAddressMisaligned` exception or an `LoadAccessFault` exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.184.3. Access

M
Always

B.184.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.184.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
```

```
misaligned");
  }
}
X[xd] = load_reserved<32>(virtual_address, aq, rl, $encoding);
```

B.184.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Extensions might perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), Read(Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        let aligned : bool =
           /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
            * to treat them as valid here; otherwise we'd need to throw an internal_error.
            */
           match width {
             BYTE   => true,
             HALF   => vaddr[0..0] == 0b0,
             WORD   => vaddr[1..0] == 0b00,
             DOUBLE => vaddr[2..0] == 0b000
           };
        /* "LR faults like a normal load, even though it's in the AMO major opcode space."
         * - Andrew Waterman, isa-dev, 10 Jul 2018.
         */
        if not(aligned)
        then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
        else match translateAddr(vaddr, Read(Data)) {
               TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
               TR_Address(addr, _) =>
                 match (width, sizeof(xlen)) {
                   (BYTE, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 1, aq, aq & rl, true), false),
                   (HALF, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 2, aq, aq & rl, true), false),
                   (WORD, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 4, aq, aq & rl, true), false),
                   (DOUBLE, 64) => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 8, aq, aq & rl, true), false),
                   _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                 }
             }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.184.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

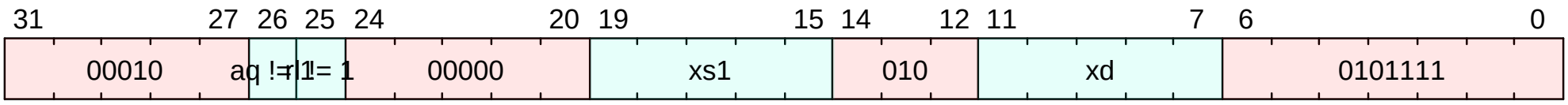
B.185. lr.w

Load reserved word

This instruction is defined by:

Zalrsc, version >= Zalrsc@1.0.0

B.185.1. Encoding



B.185.2. Description

Loads a word from the address in xs1, places the sign-extended value in xd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

<%- if MXLEN == 64 -%> The 32-bit load result is sign-extended to 64-bits. <%- end -%>

The address in xs1 must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words).

If the address is not naturally aligned, a `LoadAddressMisaligned` exception or an `LoadAccessFault` exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.185.3. Access

M
Always

B.185.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.185.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
```

```
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
  }
}
XReg load_value = load_reserved<32>(virtual_address, aq, rl, $encoding);
if (xlen() == 64) {
  X[xd] = load_value;
} else {
  X[xd] = sext(load_value[31:0], 32);
}
```

B.185.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Extensions might perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), Read(Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        let aligned : bool =
        /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
         * to treat them as valid here; otherwise we'd need to throw an internal_error.
         */
        match width {
          BYTE   => true,
          HALF   => vaddr[0..0] == 0b0,
          WORD   => vaddr[1..0] == 0b00,
          DOUBLE => vaddr[2..0] == 0b000
        };
      /* "LR faults like a normal load, even though it's in the AMO major opcode space."
       * - Andrew Waterman, isa-dev, 10 Jul 2018.
       */
      if not(aligned)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
             TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
             TR_Address(addr, _) =>
               match (width, sizeof(xlen)) {
                 (BYTE, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 1, aq, aq & rl, true), false),
                 (HALF, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 2, aq, aq & rl, true), false),
                 (WORD, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 4, aq, aq & rl, true), false),
                 (DOUBLE, 64) => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 8, aq, aq & rl, true), false),
                 _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
               }
           }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.185.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

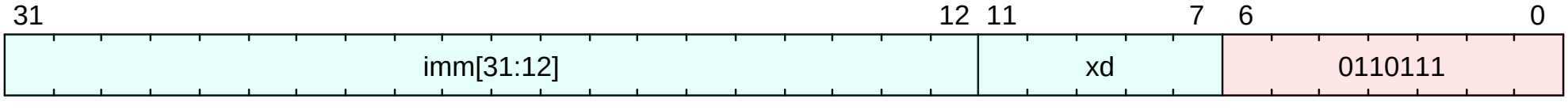
B.186. lui

Load upper immediate

This instruction is defined by:

I, version >= I@2.1.0

B.186.1. Encoding



B.186.2. Description

Load the zero-extended imm into xd.

B.186.3. Access

M
Always

B.186.4. Decode Variables

```
Bits<32> imm = {$encoding[31:12], 12'd0};
Bits<5> xd = $encoding[11:7];
```

B.186.5. IDL Operation

```
X[xd] = imm;
```

B.186.6. Sail Operation

```
{
  let off : xlenbits = sign_extend(imm @ 0x000);
  let ret : xlenbits = match op {
    RISCV_LUI   => off,
    RISCV_AUIPC => get_arch_pc() + off
  };
  X(xd) = ret;
  RETIRE_SUCCESS
}
```

B.186.7. Exceptions

This instruction does not generate synchronous exceptions.

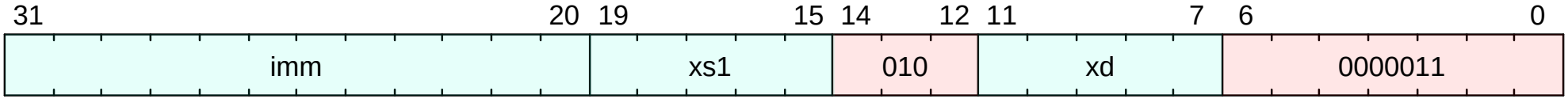
B.187. lw

Load word

This instruction is defined by:

I, version >= I@2.1.0

B.187.1. Encoding



B.187.2. Description

Load 32 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Sign extend the result.

B.187.3. Access

M
Always

B.187.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.187.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = $signed(read_memory<32>(virtual_address, $encoding));
```

B.187.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.187.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

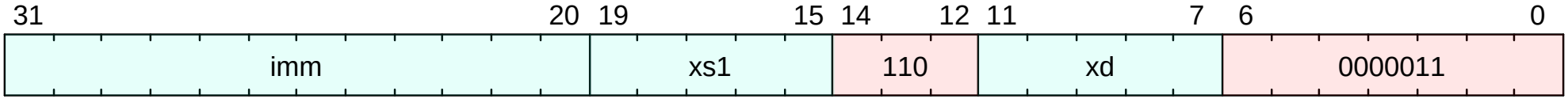
B.188. lwu

Load word unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.188.1. Encoding



B.188.2. Description

Load 64 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Zero extend the result.

B.188.3. Access

M
Always

B.188.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.188.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = read_memory<32>(virtual_address, $encoding);
```

B.188.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

B.188.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

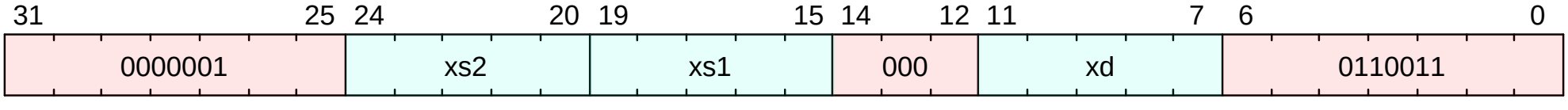
B.189. mul

Signed multiply

This instruction is defined by:

anyOf: * M, version >= M@2.0.0 * Zmmul, version >= Zmmul@1.0.0

B.189.1. Encoding



B.189.2. Description

MUL performs an XLEN-bitxXLEN-bit multiplication of `xs1` by `xs2` and places the lower XLEN bits in the destination register. Any overflow is thrown away.

If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] xdh, xs1, xs2; MUL xdl, xs1, xs2 (source register specifiers must be in same order and xdh cannot be the same as xs1 or xs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

B.189.3. Access

M
Always

B.189.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.189.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[xs1];
XReg src2 = X[xs2];
X[xd] = (src1 * src2)[MXLEN - 1:0];
```

B.189.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.189.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

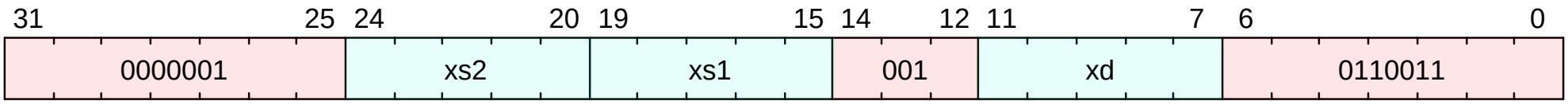
B.190. mulh

Signed multiply high

This instruction is defined by:

anyOf: * M, version >= M@2.0.0 * Zmmul, version >= Zmmul@1.0.0

B.190.1. Encoding



B.190.2. Description

Multiply the signed values in xs1 to xs2, and store the upper half of the result in xd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
mulh xdh, xs1, xs2
mul  xdl, xs1, xs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.190.3. Access

M
Always

B.190.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.190.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<1> xs1_sign_bit = X[xs1][xlen() - 1];
Bits<MXLEN `* 2> src1 = {{xlen(){xs1_sign_bit}}, X[xs1]};
Bits<1> xs2_sign_bit = X[xs2][xlen() - 1];
Bits<MXLEN `* 2> src2 = {{xlen(){xs2_sign_bit}}, X[xs2]};
X[xd] = (src1 * src2)[(xlen() * 8'd2) - 1:xlen()];
```

B.190.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
```

```
}
```

B.190.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

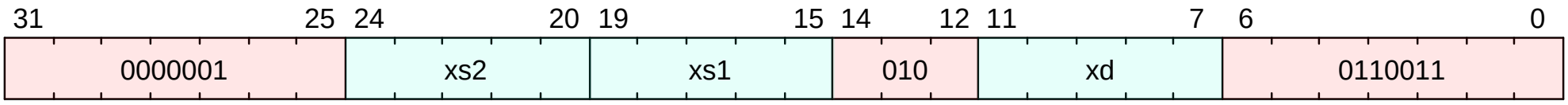
B.191. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

anyOf: * M, version >= M@2.0.0 * Zmmul, version >= Zmmul@1.0.0

B.191.1. Encoding



B.191.2. Description

Multiply the signed value in xs1 by the unsigned value in xs2, and store the upper half of the result in xd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
mulhsu xdh, xs1, xs2
mul    xdl, xs1, xs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.191.3. Access

M
Always

B.191.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.191.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<1> xs1_sign_bit = X[xs1][MXLEN - 1];
Bits<MXLEN * 8'd2> src1 = {{MXLEN{xs1_sign_bit}}, X[xs1]};
Bits<MXLEN * 8'd2> src2 = {{MXLEN{1'b0}}, X[xs2]};
X[xd] = (src1 * src2)[(MXLEN * 8'd2) - 1:MXLEN];
```

B.191.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
```

```
}
```

B.191.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

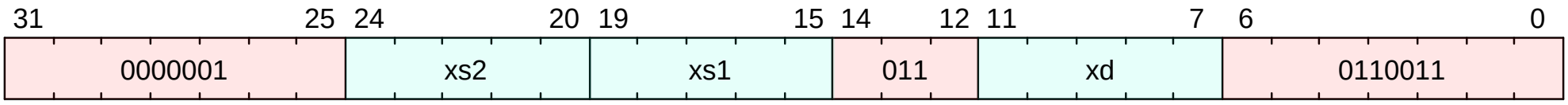
B.192. mulhu

Unsigned multiply high

This instruction is defined by:

anyOf: * M, version >= M@2.0.0 * Zmmul, version >= Zmmul@1.0.0

B.192.1. Encoding



B.192.2. Description

Multiply the unsigned values in xs1 to xs2, and store the upper half of the result in xd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
mulhu xdh, xs1, xs2
mul   xdl, xs1, xs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.192.3. Access

M
Always

B.192.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.192.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<MXLEN * 8'd2> src1 = {{MXLEN{1'b0}}, X[xs1]};
Bits<MXLEN * 8'd2> src2 = {{MXLEN{1'b0}}, X[xs2]};
X[xd] = (src1 * src2)[(MXLEN * 8'd2) - 1:MXLEN];
```

B.192.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.192.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

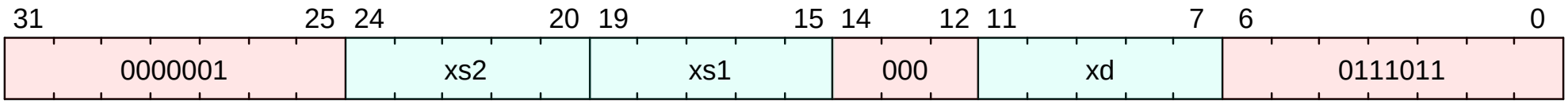
B.193. mulw

Signed 32-bit multiply

This instruction is defined by:

anyOf: * M, version >= M@2.0.0 * Zmmul, version >= Zmmul@1.0.0

B.193.1. Encoding



B.193.2. Description

Multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register.

Any overflow is thrown away.

In RV64, MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear. If the arguments are not known to be sign- or zero-extended, an alternative is to shift both arguments left by 32 bits, then use MULH[[S]U].

B.193.3. Access

M
Always

B.193.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.193.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[xs1][31:0];
Bits<32> src2 = X[xs2][31:0];
Bits<32> result = src1 * src2;
Bits<1> sign_bit = result[31];
X[xd] = {{32{sign_bit}}, result};
```

B.193.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = signed(rs1_val);
    let rs2_int : int = signed(rs2_val);
    /* to_bits requires expansion to 64 bits followed by truncation */
    let result32 = to_bits(64, rs1_int * rs2_int)[31..0];
    let result : xlenbits = sign_extend(result32);
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.193.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

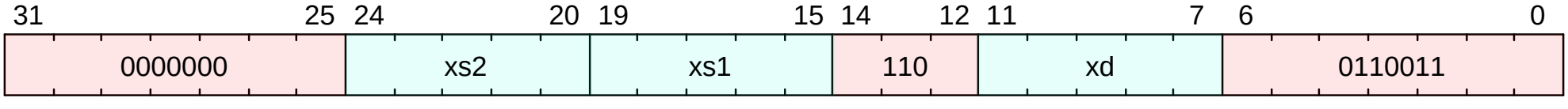
B.194. or

Or

This instruction is defined by:

I, version >= I@2.1.0

B.194.1. Encoding



B.194.2. Description

Or xs1 with xs2, and store the result in xd

B.194.3. Access

M
Always

B.194.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.194.5. IDL Operation

```
X[xd] = X[xs1] | X[xs2];
```

B.194.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.194.7. Exceptions

This instruction does not generate synchronous exceptions.

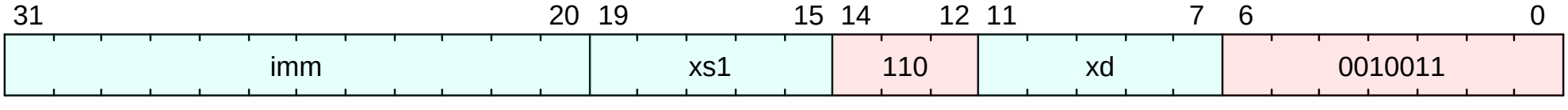
B.195. ori

Or immediate

This instruction is defined by:

I, version >= I@2.1.0

B.195.1. Encoding



B.195.2. Description

Or an immediate to the value in xs1, and store the result in xd

B.195.3. Access

M
Always

B.195.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.195.5. IDL Operation

```
X[xd] = X[xs1] | $signed(imm);
```

B.195.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => xs1_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(xs1_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(xs1_val <_u immext)),
    RISCV_ANDI  => xs1_val & immext,
    RISCV_ORI   => xs1_val | immext,
    RISCV_XORI  => xs1_val ^ immext
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.195.7. Exceptions

This instruction does not generate synchronous exceptions.

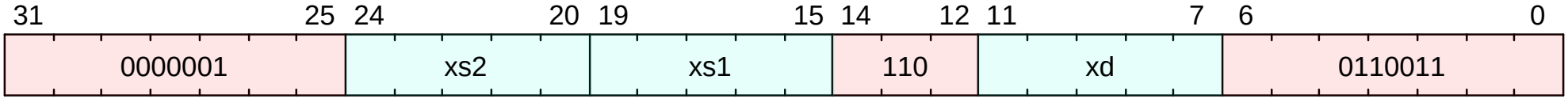
B.196. rem

Signed remainder

This instruction is defined by:

M, version >= M@2.0.0

B.196.1. Encoding



B.196.2. Description

Calculate the remainder of signed division of xs1 by xs2, and store the result in xd.

If the value in register xs2 is zero, write the value in xs1 into xd;

If the result of the division overflows, write zero into xd;

B.196.3. Access

M
Always

B.196.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.196.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[xs1];
XReg src2 = X[xs2];
if (src2 == 0) {
  X[xd] = src1;
} else if ((src1 == {1'b1, {MXLEN - 1{1'b0}}}) && (src2 == {MXLEN{1'b1}})) {
  X[xd] = 0;
} else {
  X[xd] = $signed(src1) % $signed(src2);
}
```

B.196.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let r : int = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);
    /* signed overflow case returns zero naturally as required due to -1 divisor */
    X(rd) = to_bits(sizeof(xlen), r);
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.196.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

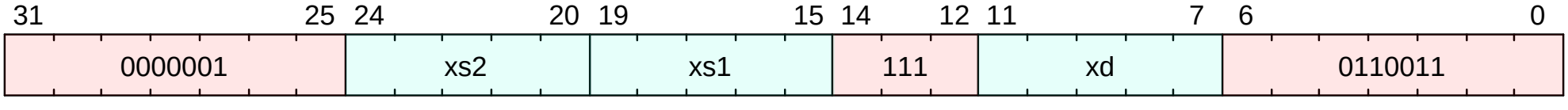
B.197. remu

Unsigned remainder

This instruction is defined by:

M, version >= M@2.0.0

B.197.1. Encoding



B.197.2. Description

Calculate the remainder of unsigned division of xs1 by xs2, and store the result in xd.

B.197.3. Access

M
Always

B.197.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.197.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[xs1];
XReg src2 = X[xs2];
if (src2 == 0) {
  X[xd] = src1;
} else {
  X[xd] = src1 % src2;
}
```

B.197.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let r : int = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);
    /* signed overflow case returns zero naturally as required due to -1 divisor */
    X(rd) = to_bits(sizeof(xlen), r);
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.197.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

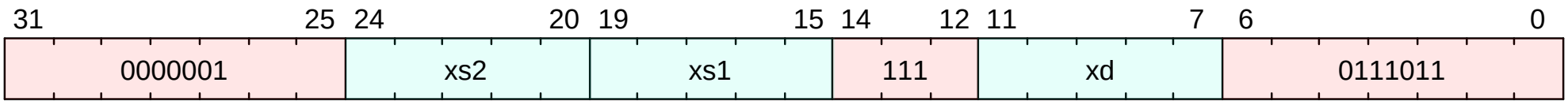
B.198. remuw

Unsigned 32-bit remainder

This instruction is defined by:

M, version >= M@2.0.0

B.198.1. Encoding



B.198.2. Description

Calculate the remainder of unsigned division of the 32-bit values in xs1 by xs2, and store the sign-extended result in xd.

If the value in xs2 is zero, xd gets the sign-extended value in xs1.

B.198.3. Access

M
Always

B.198.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.198.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[xs1][31:0];
Bits<32> src2 = X[xs2][31:0];
if (src2 == 0) {
  Bits<1> sign_bit = src1[31];
  X[xd] = {{32{sign_bit}}, src1};
} else {
  Bits<32> result = src1 % src2;
  Bits<1> sign_bit = result[31];
  X[xd] = {{32{sign_bit}}, result};
}
```

B.198.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let r : int = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);
    /* signed overflow case returns zero naturally as required due to -1 divisor */
    X(rd) = sign_extend(to_bits(32, r));
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.198.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

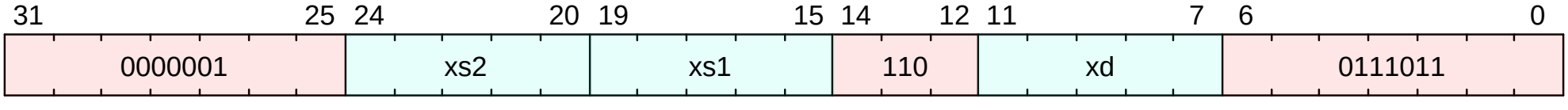
B.199. remw

Signed 32-bit remainder

This instruction is defined by:

M, version >= M@2.0.0

B.199.1. Encoding



B.199.2. Description

Calculate the remainder of signed division of the 32-bit values xs1 by xs2, and store the sign-extended result in xd.

If the value in register xs2 is zero, write the sign-extended 32-bit value in xs1 into xd;

If the result of the division overflows, write zero into xd;

B.199.3. Access

M
Always

B.199.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.199.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[xs1][31:0];
Bits<32> src2 = X[xs2][31:0];
if (src2 == 0) {
  Bits<1> sign_bit = src1[31];
  X[xd] = {{32{sign_bit}}, src1};
} else if ((src1 == {33'b1, 31'b0}) && (src2 == 32'b1)) {
  X[xd] = 0;
} else {
  Bits<32> result = $signed(src1) % $signed(src2);
  Bits<1> sign_bit = result[31];
  X[xd] = {{32{sign_bit}}, result};
}
```

B.199.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let r : int = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);
    /* signed overflow case returns zero naturally as required due to -1 divisor */
    X(rd) = sign_extend(to_bits(32, r));
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.199.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

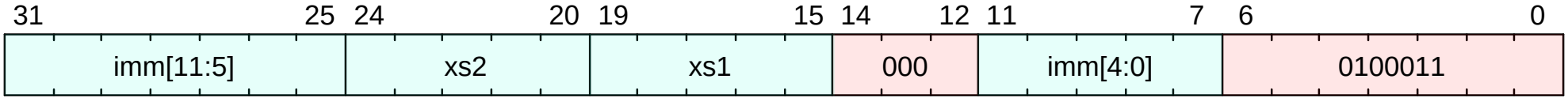
B.200. sb

Store byte

This instruction is defined by:

I, version >= I@2.1.0

B.200.1. Encoding



B.200.2. Description

Store 8 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.200.3. Access

M
Always

B.200.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.200.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<8>(virtual_address, X[xs2][7:0], $encoding);
```

B.200.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Write(Data)) {
        TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) => {
          let eares : MemoryOpResult(unit) = match width {
            BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
            HALF   => mem_write_ea(paddr, 2, aq, rl, false),
            WORD   => mem_write_ea(paddr, 4, aq, rl, false),
            DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let xs2_val = X(xs2);
              let res : MemoryOpResult(bool) = match (width) {
                BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
                HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
                WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
                DOUBLE if sizeof(xlen) >= 64
                     => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
                _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
              };
              match (res) {
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
```

```
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
    }
  }
```

B.200.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

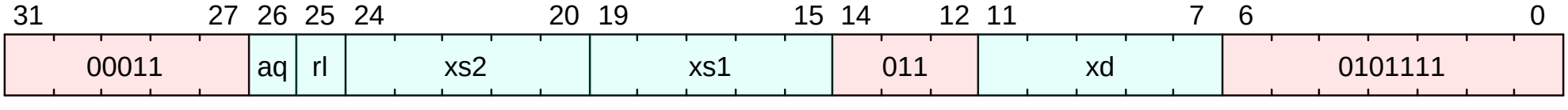
B.201. sc.d

Store conditional doubleword

This instruction is defined by:

Zalrsc, version >= Zalrsc@1.0.0

B.201.1. Encoding



B.201.2. Description

sc.d conditionally writes a doubleword in *xs2* to the address in *xs1*: the sc.d succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.d succeeds, the instruction writes the doubleword in *xs2* to memory, and it writes zero to *xd*. If the sc.d fails, the instruction does not write to memory, and it writes a nonzero value to *xd*. For the purposes of memory protection, a failed sc.d may be treated like a store. Regardless of success or failure, executing an sc.d instruction invalidates any reservation held by this hart.

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in *xs1* must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- during a preemptive context switch, and
- if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.

The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.

If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.201.3. Access

M
Always

B.201.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.201.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
XReg value = X[xs2];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
  }
}
Boolean success = store_conditional<64>(virtual_address, value, aq, rl, $encoding);
X[xd] = success ? 0 : 1;
```

B.201.6. Sail Operation

```
{
  if speculate_conditional () == false then {
    /* should only happen in rmem
     * rmem: allow SC to fail very early
     */
    X(rd) = zero_extend(0b1); RETIRE_SUCCESS
  } else {
    if extension("A") then {
      /* normal non-rmem case
       * rmem: SC is allowed to succeed (but might fail later)
       */
      /* Get the address, X(rs1) (no offset).
       * Extensions might perform additional checks on address validity.
       */
      match ext_data_get_addr(rs1, zeros(), Write(Data), width) {
        Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
        Ext_DataAddr_OK(vaddr) => {
          let aligned : bool =
            /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
             * to treat them as valid here; otherwise we'd need to throw an internal_error.
             */
```

```
  match width {
    BYTE   => true,
    HALF   => vaddr[0..0] == 0b0,
    WORD   => vaddr[1..0] == 0b00,
    DOUBLE => vaddr[2..0] == 0b000
  };
if not(aligned)
then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
else {
  if match_reservation(vaddr) == false then {
    /* cannot happen in rmem */
    X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS
  } else {
    match translateAddr(vaddr, Write(Data)) {  /* Write and ReadWrite are equivalent here:
                                                * both result in a SAMO exception */
      TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      TR_Address(addr, _) => {
        let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
          (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
          (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
          (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
          (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
          _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            rs2_val = X(rs2);
            let res : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
              (BYTE, _)    => mem_write_value(addr, 1, rs2_val[7..0], aq & rl, rl, true),
              (HALF, _)    => mem_write_value(addr, 2, rs2_val[15..0], aq & rl, rl, true),
              (WORD, _)    => mem_write_value(addr, 4, rs2_val[31..0], aq & rl, rl, true),
              (DOUBLE, 64) => mem_write_value(addr, 8, rs2_val,        aq & rl, rl, true),
              _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
            };
            match (res) {
              MemValue(true)  => { X(rd) = zero_extend(0b0); cancel_reservation(); RETIRE_SUCCESS },
              MemValue(false) => { X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS },
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
            }
          }
        }
      }
    }
  }
} else {
  handle_illegal();
  RETIRE_FAIL
}
```

B.201.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

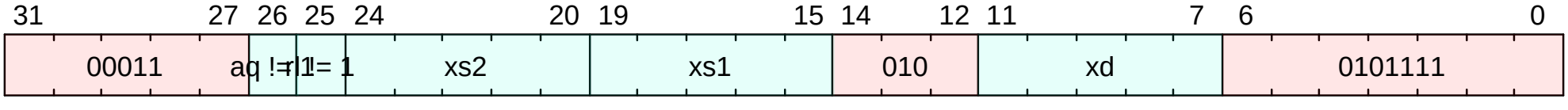
B.202. sc.w

Store conditional word

This instruction is defined by:

Zalrsc, version >= Zalrsc@1.0.0

B.202.1. Encoding



B.202.2. Description

sc.w conditionally writes a word in *xs2* to the address in *xs1*: the sc.w succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.w succeeds, the instruction writes the word in *xs2* to memory, and it writes zero to *xd*. If the sc.w fails, the instruction does not write to memory, and it writes a nonzero value to *xd*. For the purposes of memory protection, a failed sc.w may be treated like a store. Regardless of success or failure, executing an sc.w instruction invalidates any reservation held by this hart.

<%- if MXLEN == 64 -%>

If a value other than 0 or 1 is defined as a result for sc.w, the value will before sign-extended into *xd*. <%- end -%>

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in *xs1* must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

Emulating misaligned LR/SC sequences is impractical in most systems.

Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

The platform should provide a means to determine the size and shape of the reservation set.

A platform specification may constrain the size and shape of the reservation set.

A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:

- during a preemptive context switch, and
- if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.

The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and

> that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

> The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.
>
> If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

B.202.3. Access

M
Always

B.202.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.202.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[xs1];
XReg value = X[xs2];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address is
misaligned");
  }
}
Boolean success = store_conditional<32>(virtual_address, value, aq, rl, $encoding);
X[xd] = success ? 0 : 1;
```

B.202.6. Sail Operation

```
{
  if speculate_conditional () == false then {
    /* should only happen in rmem
     * rmem: allow SC to fail very early
     */
    X(rd) = zero_extend(0b1); RETIRE_SUCCESS
  } else {
    if extension("A") then {
      /* normal non-rmem case
       * rmem: SC is allowed to succeed (but might fail later)
       */
      /* Get the address, X(rs1) (no offset).
       * Extensions might perform additional checks on address validity.
       */
      match ext_data_get_addr(rs1, zeros(), Write(Data), width) {
```

```
        Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
        Ext_DataAddr_OK(vaddr) => {
          let aligned : bool =
            /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
             * to treat them as valid here; otherwise we'd need to throw an internal_error.
             */
            match width {
              BYTE   => true,
              HALF   => vaddr[0..0] == 0b0,
              WORD   => vaddr[1..0] == 0b00,
              DOUBLE => vaddr[2..0] == 0b000
            };
          if not(aligned)
          then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
          else {
            if match_reservation(vaddr) == false then {
              /* cannot happen in rmem */
              X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS
            } else {
              match translateAddr(vaddr, Write(Data)) {  /* Write and ReadWrite are equivalent here:
                                                          * both result in a SAMO exception */
                TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                TR_Address(addr, _) => {
                  let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
                    (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
                    (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
                    (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
                    (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
                    _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
                  };
                  match (eares) {
                    MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                    MemValue(_) => {
                      rs2_val = X(rs2);
                      let res : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                        (BYTE, _)    => mem_write_value(addr, 1, rs2_val[7..0], aq & rl, rl, true),
                        (HALF, _)    => mem_write_value(addr, 2, rs2_val[15..0], aq & rl, rl, true),
                        (WORD, _)    => mem_write_value(addr, 4, rs2_val[31..0], aq & rl, rl, true),
                        (DOUBLE, 64) => mem_write_value(addr, 8, rs2_val,        aq & rl, rl, true),
                        _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
                      };
                      match (res) {
                        MemValue(true)  => { X(rd) = zero_extend(0b0); cancel_reservation(); RETIRE_SUCCESS },
                        MemValue(false) => { X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS },
                        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                      }
                    }
                  }
                }
              }
            } else {
              handle_illegal();
              RETIRE_FAIL
            }
          }
        }
```

B.202.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

B.203. sd

Store doubleword

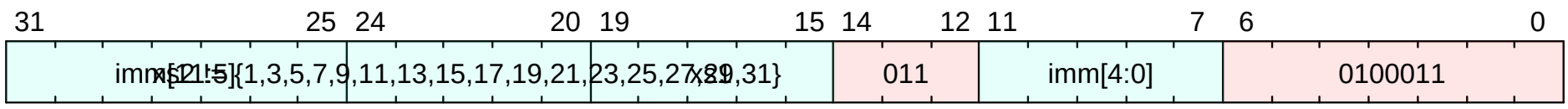
This instruction is defined by:

anyOf: * I, version >= I@2.1.0 * Zilsd, version >= Zilsd@1.0.0

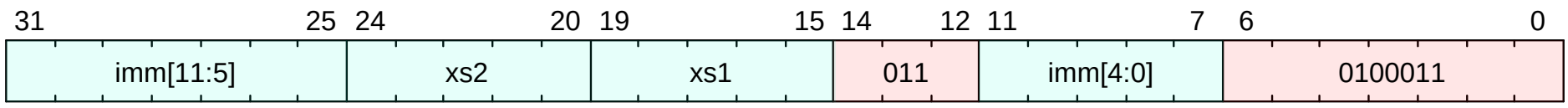
B.203.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.203.2. Description

For RV64, store 64 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset. <% if ext?(:Zilsd) %> For RV32, store doubleword from even/odd register pair. <% end %>

B.203.3. Access

M
Always

B.203.4. Decode Variables

RV32

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

RV64

```
signed Bits<12> imm = sext({$encoding[31:25], $encoding[11:7]});
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.203.5. IDL Operation

```
Bits<64> data;
XReg virtual_address = X[xs1] + $signed(imm);
if (xlen() == 32) {
  if (implemented?(ExtensionName::Zclsd)) {
    data = {X[xs2 + 1], X[xs2]};
  } else {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else {
  data = X[xs2];
}
write_memory<64>(virtual_address, data, $encoding);
```

B.203.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
```

```
/* Get the address, X(xs1) + offset.
   Some extensions perform additional checks on address validity. */
match ext_data_get_addr(xs1, offset, Write(Data), width) {
  Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
  Ext_DataAddr_OK(vaddr) =>
    if   check_misaligned(vaddr, width)
    then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
    else match translateAddr(vaddr, Write(Data)) {
      TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      TR_Address(paddr, _) => {
        let eares : MemoryOpResult(unit) = match width {
          BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
          HALF   => mem_write_ea(paddr, 2, aq, rl, false),
          WORD   => mem_write_ea(paddr, 4, aq, rl, false),
          DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let xs2_val = X(xs2);
            let res : MemoryOpResult(bool) = match (width) {
              BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
              HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
              WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
              DOUBLE if sizeof(xlen) >= 64
                   => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
              _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
            };
            match (res) {
              MemValue(true)  => RETIRE_SUCCESS,
              MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
              MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
            }
          }
        }
      }
    }
  }
}
```

B.203.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

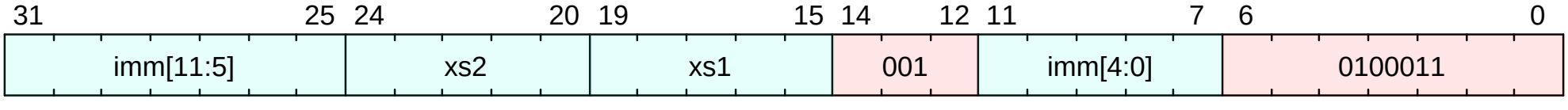
B.204. sh

Store halfword

This instruction is defined by:

I, version >= I@2.1.0

B.204.1. Encoding



B.204.2. Description

Store 16 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.204.3. Access

M
Always

B.204.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.204.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<16>(virtual_address, X[xs2][15:0], $encoding);
```

B.204.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Write(Data)) {
        TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) => {
          let eares : MemoryOpResult(unit) = match width {
            BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
            HALF   => mem_write_ea(paddr, 2, aq, rl, false),
            WORD   => mem_write_ea(paddr, 4, aq, rl, false),
            DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let xs2_val = X(xs2);
              let res : MemoryOpResult(bool) = match (width) {
                BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
                HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
                WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
                DOUBLE if sizeof(xlen) >= 64
                     => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
                _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
              };
              match (res) {
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
```

```
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
    }
  }
```

B.204.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

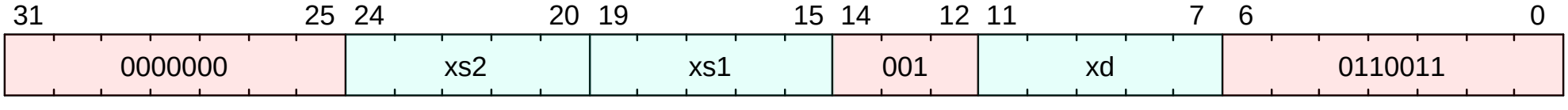
B.205. sll

Shift left logical

This instruction is defined by:

I, version >= I@2.1.0

B.205.1. Encoding



B.205.2. Description

Shift the value in xs1 left by the value in the lower 6 bits of xs2, and store the result in xd.

B.205.3. Access

M
Always

B.205.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.205.5. IDL Operation

```
if (xlen() == 64) {
  X[xd] = X[xs1] << X[xs2][5:0];
} else {
  X[xd] = X[xs1] << X[xs2][4:0];
}
```

B.205.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.205.7. Exceptions

This instruction does not generate synchronous exceptions.

B.206. slli

Shift left logical immediate

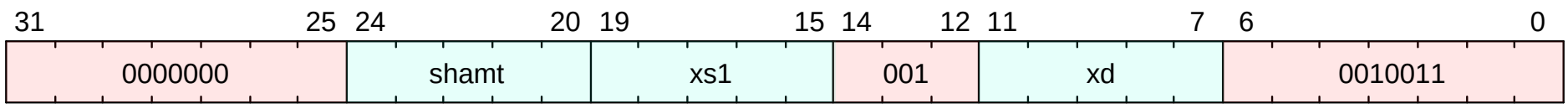
This instruction is defined by:

I, version >= I@2.1.0

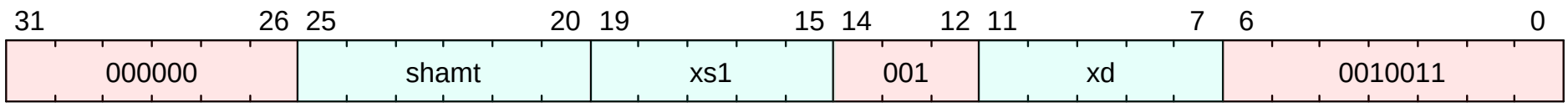
B.206.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.206.2. Description

Shift the value in xs1 left by shamt, and store the result in xd

B.206.3. Access

M
Always

B.206.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.206.5. IDL Operation

```
X[xd] = X[xs1] << shamt;
```

B.206.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then xs1_val << shamt[4..0]
                  else xs1_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then xs1_val >> shamt[4..0]
                  else xs1_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, shamt[4..0])
                  else shift_right_arith64(xs1_val, shamt)
  };
```

```
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.206.7. Exceptions

This instruction does not generate synchronous exceptions.

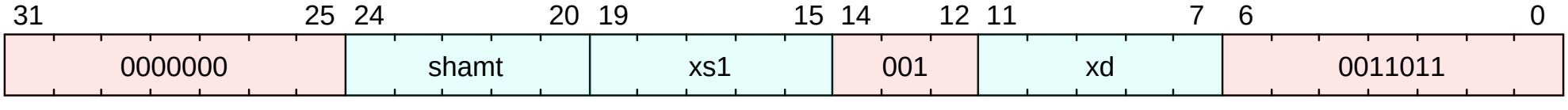
B.207. slliw

Shift left logical immediate word

This instruction is defined by:

I, version >= I@2.1.0

B.207.1. Encoding



B.207.2. Description

Shift the 32-bit value in xs1 left by shamt, and store the sign-extended result in xd

B.207.3. Access

M
Always

B.207.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.207.5. IDL Operation

```
X[xd] = sext(X[xs1] << shamt, 31);
```

B.207.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.207.7. Exceptions

This instruction does not generate synchronous exceptions.

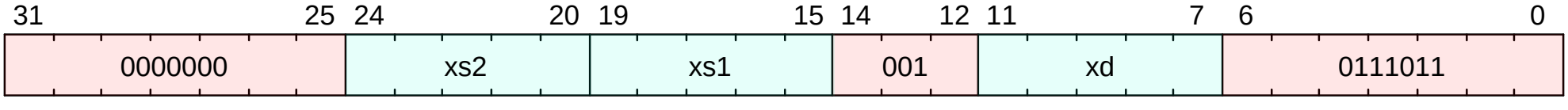
B.208. sllw

Shift left logical word

This instruction is defined by:

I, version >= I@2.1.0

B.208.1. Encoding



B.208.2. Description

Shift the 32-bit value in `xs1` left by the value in the lower 5 bits of `xs2`, and store the sign-extended result in `xd`.

B.208.3. Access

M
Always

B.208.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.208.5. IDL Operation

```
X[xd] = sext(X[xs1] << X[xs2][4:0], 31);
```

B.208.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISCV_ADDW => xs1_val + xs2_val,
    RISCV_SUBW => xs1_val - xs2_val,
    RISCV_SLLW => xs1_val << (xs2_val[4..0]),
    RISCV_SRLW => xs1_val >> (xs2_val[4..0]),
    RISCV_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.208.7. Exceptions

This instruction does not generate synchronous exceptions.

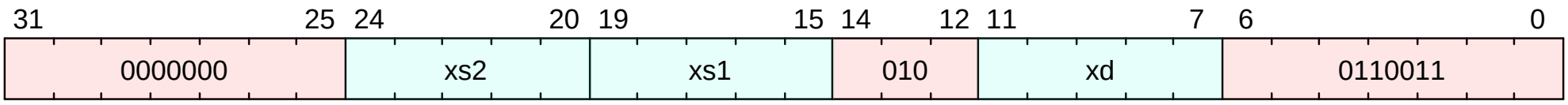
B.209. slt

Set on less than

This instruction is defined by:

I, version >= I@2.1.0

B.209.1. Encoding



B.209.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the value in register `xs2`, where both sources are treated as signed numbers, else 0 is written to `xd`.

B.209.3. Access

M
Always

B.209.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.209.5. IDL Operation

```
XReg src1 = X[xs1];
XReg src2 = X[xs2];
X[xd] = ($signed(src1) < $signed(src2)) ? '1 : '0;
```

B.209.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.209.7. Exceptions

This instruction does not generate synchronous exceptions.

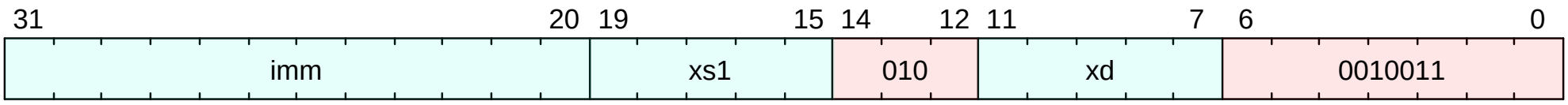
B.210. slti

Set on less than immediate

This instruction is defined by:

I, version >= I@2.1.0

B.210.1. Encoding



B.210.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to `xd`.

B.210.3. Access

M
Always

B.210.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.210.5. IDL Operation

```
X[xd] = ($signed(X[xs1]) < $signed(imm)) ? '1 : '0;
```

B.210.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => xs1_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(xs1_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(xs1_val <_u immext)),
    RISCV_ANDI  => xs1_val & immext,
    RISCV_ORI   => xs1_val | immext,
    RISCV_XORI  => xs1_val ^ immext
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.210.7. Exceptions

This instruction does not generate synchronous exceptions.

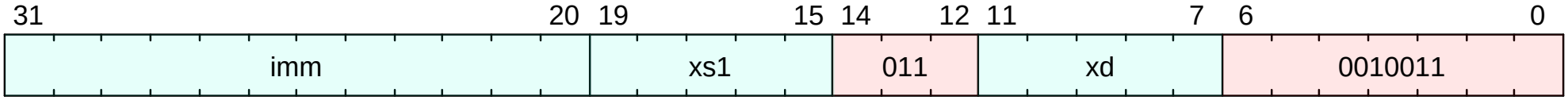
B.211. sltiu

Set on less than immediate unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.211.1. Encoding



B.211.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the sign-extended immediate when both are treated as unsigned numbers (*i.e.*, the immediate is first sign-extended to XLEN bits then treated as an unsigned number), else 0 is written to `xd`.

`sltiu xd, xs1, 1` sets `xd` to 1 if `xs1` equals zero, otherwise sets `xd` to 0 (assembler pseudoinstruction `SEQZ xd, rs`).

B.211.3. Access

M
Always

B.211.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.211.5. IDL Operation

```
Bits<MXLEN> sign_extend_imm = $signed(imm);
X[xd] = (X[xs1] < sign_extend_imm) ? 1 : 0;
```

B.211.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => xs1_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(xs1_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(xs1_val <_u immext)),
    RISCV_ANDI  => xs1_val & immext,
    RISCV_ORI   => xs1_val | immext,
    RISCV_XORI  => xs1_val ^ immext
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.211.7. Exceptions

This instruction does not generate synchronous exceptions.

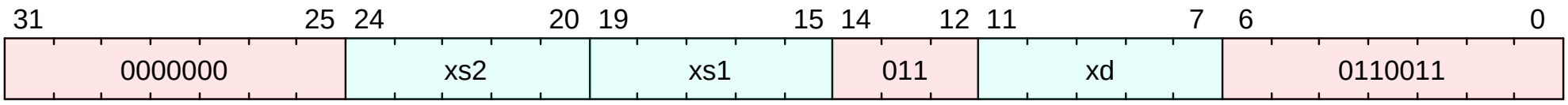
B.212. sltu

Set on less than unsigned

This instruction is defined by:

I, version >= I@2.1.0

B.212.1. Encoding



B.212.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the value in register `xs2`, where both sources are treated as unsigned numbers, else 0 is written to `xd`.

B.212.3. Access

M
Always

B.212.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.212.5. IDL Operation

```
X[xd] = (X[xs1] < X[xs2]) ? 1 : 0;
```

B.212.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.212.7. Exceptions

This instruction does not generate synchronous exceptions.

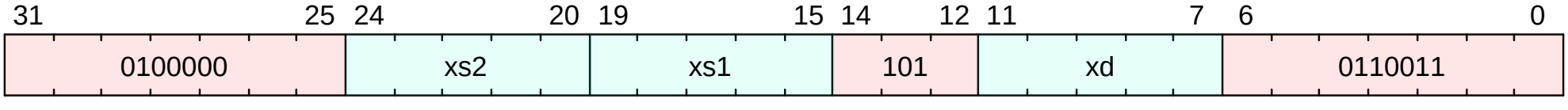
B.213. sra

Shift right arithmetic

This instruction is defined by:

I, version >= I@2.1.0

B.213.1. Encoding



B.213.2. Description

Arithmetic shift the value in `xs1` right by the value in the lower 5 bits of `xs2`, and store the result in `xd`.

B.213.3. Access

M
Always

B.213.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.213.5. IDL Operation

```
if (xlen() == 64) {
  X[xd] = X[xs1] >>> X[xs2][5:0];
} else {
  X[xd] = X[xs1] >>> X[xs2][4:0];
}
```

B.213.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.213.7. Exceptions

This instruction does not generate synchronous exceptions.

B.214. srai

Shift right arithmetic immediate

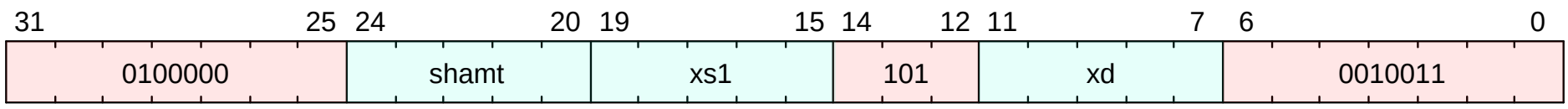
This instruction is defined by:

I, version >= I@2.1.0

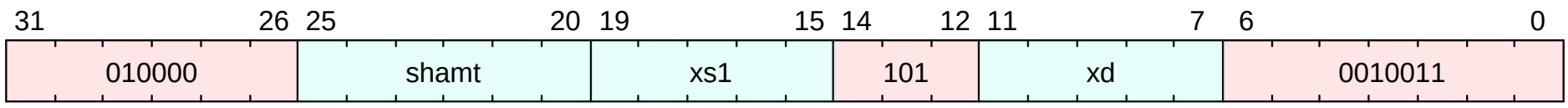
B.214.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.214.2. Description

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the value in xs1 right by shamt, and store the result in xd.

B.214.3. Access

M
Always

B.214.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.214.5. IDL Operation

```
X[xd] = X[xs1] >>> shamt;
```

B.214.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then xs1_val << shamt[4..0]
                  else xs1_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then xs1_val >> shamt[4..0]
                  else xs1_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, shamt[4..0])
                  else shift_right_arith64(xs1_val, shamt)
  };
```

```
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.214.7. Exceptions

This instruction does not generate synchronous exceptions.

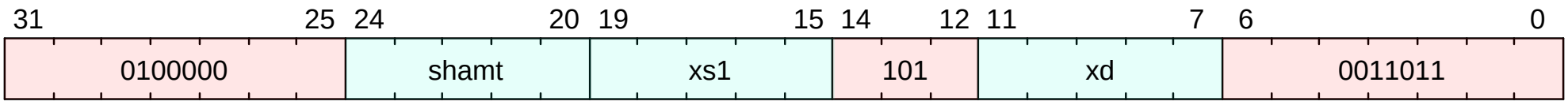
B.215. sraiw

Shift right arithmetic immediate word

This instruction is defined by:

I, version >= I@2.1.0

B.215.1. Encoding



B.215.2. Description

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the 32-bit value in xs1 right by shamt, and store the sign-extended result in xd.

B.215.3. Access

M
Always

B.215.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.215.5. IDL Operation

```
XReg operand = sext(X[xs1], 31);
X[xd] = sext(operand >>> shamt, 31);
```

B.215.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.215.7. Exceptions

This instruction does not generate synchronous exceptions.

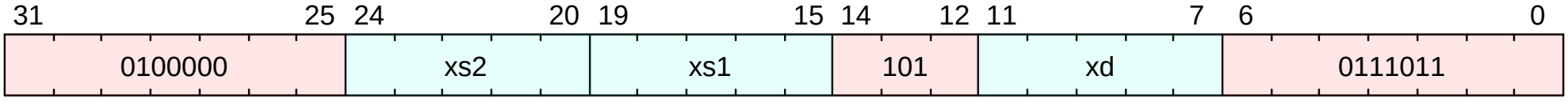
B.216. sraw

Shift right arithmetic word

This instruction is defined by:

I, version >= I@2.1.0

B.216.1. Encoding



B.216.2. Description

Arithmetic shift the 32-bit value in `xs1` right by the value in the lower 5 bits of `xs2`, and store the sign-extended result in `xd`.

B.216.3. Access

M
Always

B.216.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.216.5. IDL Operation

```
XReg operand1 = sext(X[xs1], 31);
X[xd] = sext(operand1 >>> X[xs2][4:0], 31);
```

B.216.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISCV_ADDW => xs1_val + xs2_val,
    RISCV_SUBW => xs1_val - xs2_val,
    RISCV_SLLW => xs1_val << (xs2_val[4..0]),
    RISCV_SRLW => xs1_val >> (xs2_val[4..0]),
    RISCV_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.216.7. Exceptions

This instruction does not generate synchronous exceptions.

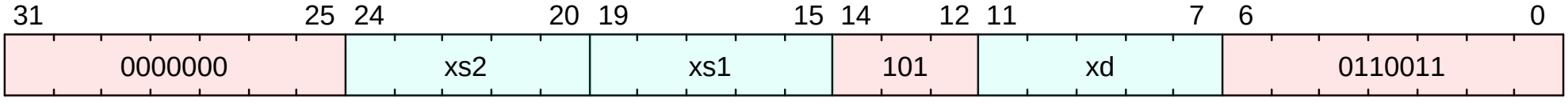
B.217. srl

Shift right logical

This instruction is defined by:

I, version >= I@2.1.0

B.217.1. Encoding



B.217.2. Description

Logical shift the value in `xs1` right by the value in the lower bits of `xs2`, and store the result in `xd`.

B.217.3. Access

M
Always

B.217.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.217.5. IDL Operation

```
if (xlen() == 64) {
  X[xd] = X[xs1] >> X[xs2][5:0];
} else {
  X[xd] = X[xs1] >> X[xs2][4:0];
}
```

B.217.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.217.7. Exceptions

This instruction does not generate synchronous exceptions.

B.218. srli

Shift right logical immediate

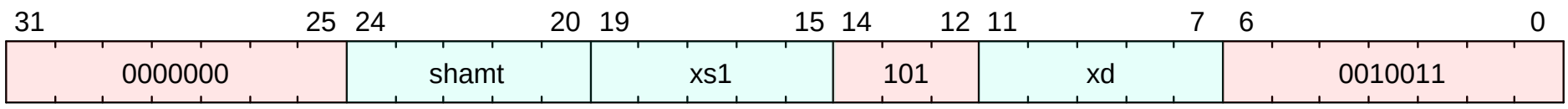
This instruction is defined by:

I, version >= I@2.1.0

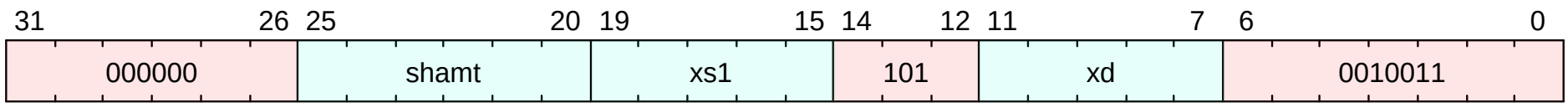
B.218.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.218.2. Description

Shift the value in xs1 right by shamt, and store the result in xd

B.218.3. Access

M
Always

B.218.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.218.5. IDL Operation

```
X[xd] = X[xs1] >> shamt;
```

B.218.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then xs1_val << shamt[4..0]
                  else xs1_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then xs1_val >> shamt[4..0]
                  else xs1_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, shamt[4..0])
                  else shift_right_arith64(xs1_val, shamt)
  };
```

```
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.218.7. Exceptions

This instruction does not generate synchronous exceptions.

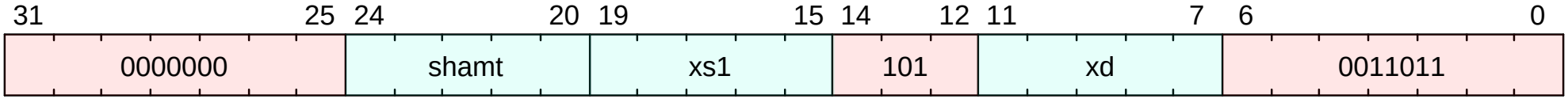
B.219. srliw

Shift right logical immediate word

This instruction is defined by:

I, version >= I@2.1.0

B.219.1. Encoding



B.219.2. Description

Shift the 32-bit value in xs1 right by shamt, and store the sign-extended result in xd

B.219.3. Access

M
Always

B.219.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.219.5. IDL Operation

```
XReg operand = X[xs1][31:0];
X[xd] = sext(operand >> shamt, 31);
```

B.219.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.219.7. Exceptions

This instruction does not generate synchronous exceptions.

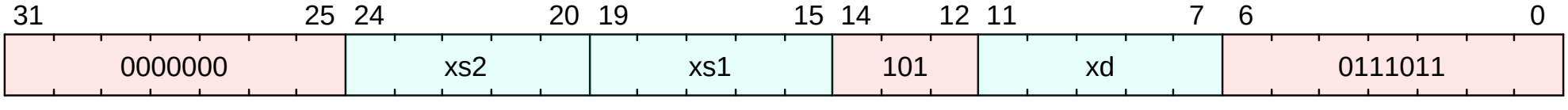
B.220. srlw

Shift right logical word

This instruction is defined by:

I, version >= I@2.1.0

B.220.1. Encoding



B.220.2. Description

Logical shift the 32-bit value in `xs1` right by the value in the lower 5 bits of `xs2`, and store the sign-extended result in `xd`.

B.220.3. Access

M
Always

B.220.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.220.5. IDL Operation

```
X[xd] = sext(X[xs1][31:0] >> X[xs2][4:0], 31);
```

B.220.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISCV_ADDW => xs1_val + xs2_val,
    RISCV_SUBW => xs1_val - xs2_val,
    RISCV_SLLW => xs1_val << (xs2_val[4..0]),
    RISCV_SRLW => xs1_val >> (xs2_val[4..0]),
    RISCV_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.220.7. Exceptions

This instruction does not generate synchronous exceptions.

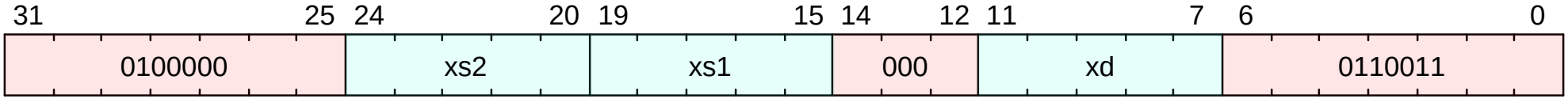
B.221. sub

Subtract

This instruction is defined by:

I, version >= I@2.1.0

B.221.1. Encoding



B.221.2. Description

Subtract the value in xs2 from xs1, and store the result in xd

B.221.3. Access

M
Always

B.221.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.221.5. IDL Operation

```
XReg t0 = X[xs1];
XReg t1 = X[xs2];
X[xd] = t0 - t1;
```

B.221.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.221.7. Exceptions

This instruction does not generate synchronous exceptions.

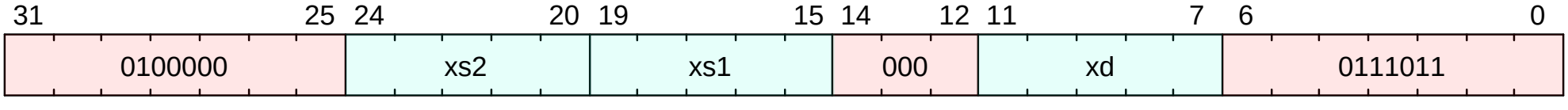
B.222. subw

Subtract word

This instruction is defined by:

I, version >= I@2.1.0

B.222.1. Encoding



B.222.2. Description

Subtract the 32-bit values in xs2 from xs1, and store the sign-extended result in xd

B.222.3. Access

M
Always

B.222.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.222.5. IDL Operation

```
Bits<32> t0 = X[xs1][31:0];
Bits<32> t1 = X[xs2][31:0];
X[xd] = sext(t0 - t1, 31);
```

B.222.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let xs2_val = (X(xs2))[31..0];
  let result : bits(32) = match op {
    RISCV_ADDW => xs1_val + xs2_val,
    RISCV_SUBW => xs1_val - xs2_val,
    RISCV_SLLW => xs1_val << (xs2_val[4..0]),
    RISCV_SRLW => xs1_val >> (xs2_val[4..0]),
    RISCV_SRAW => shift_right_arith32(xs1_val, xs2_val[4..0])
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.222.7. Exceptions

This instruction does not generate synchronous exceptions.

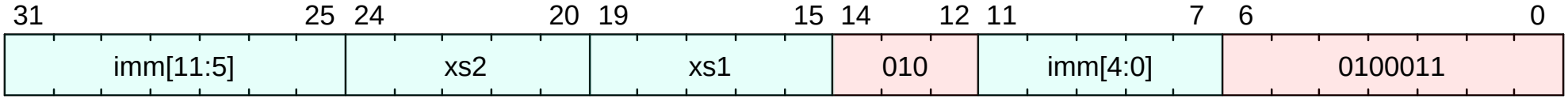
B.223. sw

Store word

This instruction is defined by:

I, version >= I@2.1.0

B.223.1. Encoding



B.223.2. Description

Store 32 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.223.3. Access

M
Always

B.223.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.223.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<32>(virtual_address, X[xs2][31:0], $encoding);
```

B.223.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Write(Data)) {
        TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) => {
          let eares : MemoryOpResult(unit) = match width {
            BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
            HALF   => mem_write_ea(paddr, 2, aq, rl, false),
            WORD   => mem_write_ea(paddr, 4, aq, rl, false),
            DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              let xs2_val = X(xs2);
              let res : MemoryOpResult(bool) = match (width) {
                BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
                HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
                WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
                DOUBLE if sizeof(xlen) >= 64
                     => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
                _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
              };
              match (res) {
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
```

```
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
  }
}
```

B.223.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

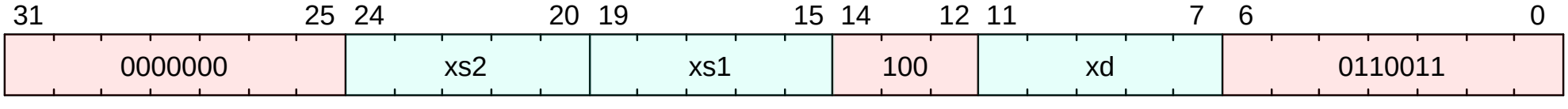
B.224. xor

Exclusive Or

This instruction is defined by:

I, version >= I@2.1.0

B.224.1. Encoding



B.224.2. Description

Exclusive or xs1 with xs2, and store the result in xd

B.224.3. Access

M
Always

B.224.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.224.5. IDL Operation

```
X[xd] = X[xs1] ^ X[xs2];
```

B.224.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let xs2_val = X(xs2);
  let result : xlenbits = match op {
    RISCV_ADD  => xs1_val + xs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(xs1_val <_s xs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(xs1_val <_u xs2_val)),
    RISCV_AND  => xs1_val & xs2_val,
    RISCV_OR   => xs1_val | xs2_val,
    RISCV_XOR  => xs1_val ^ xs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then xs1_val << (xs2_val[4..0])
                  else xs1_val << (xs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then xs1_val >> (xs2_val[4..0])
                  else xs1_val >> (xs2_val[5..0]),
    RISCV_SUB  => xs1_val - xs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, xs2_val[4..0])
                  else shift_right_arith64(xs1_val, xs2_val[5..0])
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.224.7. Exceptions

This instruction does not generate synchronous exceptions.

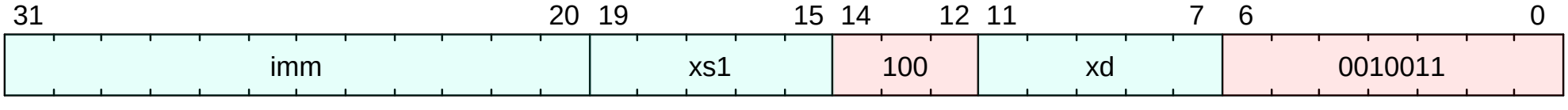
B.225. xori

Exclusive Or immediate

This instruction is defined by:

I, version >= I@2.1.0

B.225.1. Encoding



B.225.2. Description

Exclusive or an immediate to the value in xs1, and store the result in xd

B.225.3. Access

M
Always

B.225.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

B.225.5. IDL Operation

```
X[xd] = X[xs1] ^ $signed(imm);
```

B.225.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => xs1_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(xs1_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(xs1_val <_u immext)),
    RISCV_ANDI  => xs1_val & immext,
    RISCV_ORI   => xs1_val | immext,
    RISCV_XORI  => xs1_val ^ immext
  };
  X(xd) = result;
  RETIRE_SUCCESS
}
```

B.225.7. Exceptions

This instruction does not generate synchronous exceptions.

Appendix C: CSR Details

C.1. cycle

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR mcycle.

Privilege mode access is controlled with mcounteren.CY, scounteren.CY, and hcounteren.CY as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.1.1. Attributes

CSR Address	0xc00
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

C.1.2. Format

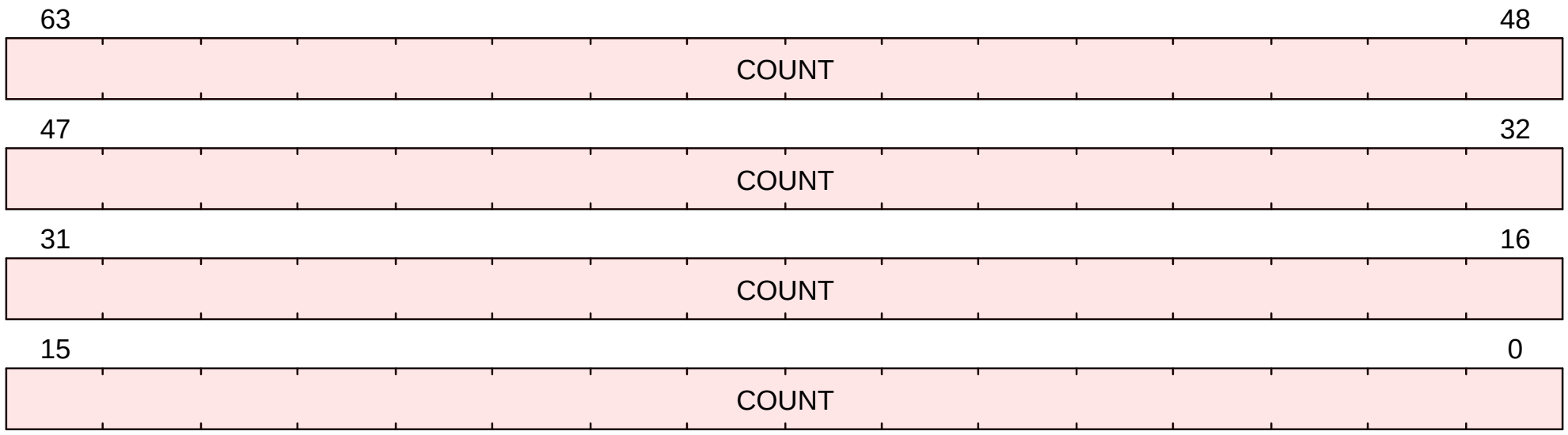


Figure 1. cycle format

C.1.3. Field Summary

Name	Location	Type	Reset Value
cycle.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.1.4. Fields

cycle.COUNT Field

Location:

63:0

Description:

Alias of mcycle.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.1.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.CY & scounteren.CY) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.CY == 1'b0 && mcounteren.CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.CY & scounteren.CY) == 1'b0) && (mcounteren.CY == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle();
```

C.2. cycleh

High-half cycle counter for RDCYCLE Instruction

cycleh is only defined in RV32.

Alias for M-mode CSR mcycleh.

Privilege mode access is controlled with mcounteren.CY, scounteren.CY, and hcounteren.CY as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

C.2.1. Attributes

CSR Address	0xc80
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

C.2.2. Format

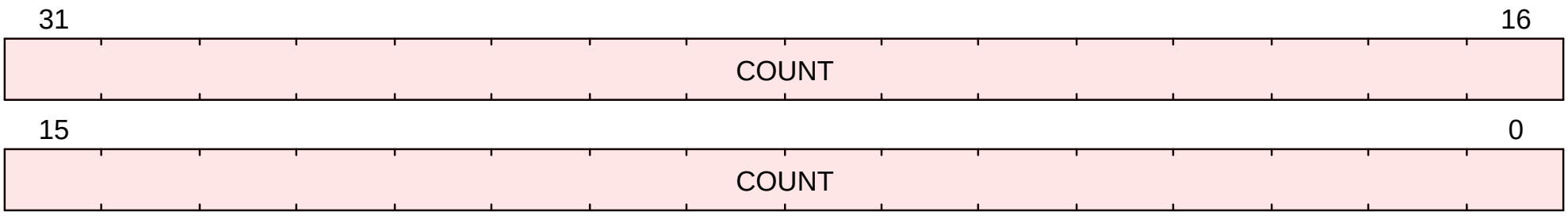


Figure 2. cycleh format

C.2.3. Field Summary

Name	Location	Type	Reset Value
cycleh.COUNT	31:0	RO-H	UNDEFINED_LEGAL

C.2.4. Fields

cycleh.COUNT Field

Location:
31:0

Description:
Alias of mcycleh.COUNT.

Type:
RO-H

Reset value:
UNDEFINED_LEGAL

C.2.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.CY & scounteren.CY) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.CY == 1'b0 && mcounteren.CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.CY & scounteren.CY) == 1'b0) && (mcounteren.CY == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle()[63:32];
```

C.3. fcsr

Floating-point control and status register (frm + fflags)

The floating-point control and status register, fcsr, is a RISC-V control and status register (CSR). It is a 32-bit read/write register that selects the dynamic rounding mode for floating-point arithmetic operations and holds the accrued exception flags, as shown in Floating-Point Control and Status Register.

Floating-point control and status register

Unresolved directive in RVI20-64-CRD.adoc - include::images/wavedrom/float-csr.adoc[]

The fcsr register can be read and written with the FRCSR and FSCSR instructions, which are assembler pseudoinstructions built on the underlying CSR access instructions. FRCSR reads fcsr by copying it into integer register *rd*. FSCSR swaps the value in fcsr by copying the original value into integer register *rd*, and then writing a new value obtained from integer register *rs1* into fcsr.

The fields within the fcsr can also be accessed individually through different CSR addresses, and separate assembler pseudoinstructions are defined for these accesses. The FRRM instruction reads the Rounding Mode field frm (fcsr bits 7—5) and copies it into the least-significant three bits of integer register *rd*, with zero in all other bits. FSRM swaps the value in frm by copying the original value into integer register *rd*, and then writing a new value obtained from the three least-significant bits of integer register *rs1* into frm. FRFLAGS and FSFLAGS are defined analogously for the Accrued Exception Flags field fflags (fcsr bits 4—0).

Bits 31—8 of the fcsr are reserved for other standard extensions. If these extensions are not present, implementations shall ignore writes to these bits and supply a zero value when read. Standard software should preserve the contents of these bits.

Floating-point operations use either a static rounding mode encoded in the instruction, or a dynamic rounding mode held in frm. Rounding modes are encoded as shown in Table 7. A value of 111 in the instruction's *rm* field selects the dynamic rounding mode held in frm. The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111). Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

The C99 language standard effectively mandates the provision of a dynamic rounding mode register. In typical implementations, writes to the dynamic rounding mode CSR state will serialize the pipeline. Static rounding modes are used to implement specialized arithmetic operations that often have to switch frequently between different rounding modes.

The ratified version of the F spec mandated that an illegal-instruction exception was raised when an instruction was executed with a reserved dynamic rounding mode. This has been weakened to reserved, which matches the behavior of static rounding-mode instructions. Raising an illegal-instruction exception is still valid behavior when encountering a reserved encoding, so implementations compatible with the ratified spec are compatible with the weakened spec.

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software, as shown in Table 8. The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

Table 14. Accrued exception flag encoding.

Flag Mnemonic	Flag Meaning
NV	Invalid Operation
DZ	Divide by Zero
OF	Overflow
UF	Underflow
NX	Inexact

As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

C.3.1. Attributes

CSR Address	0x3
Defining extension	F, version >= F@2.2.0
Length	32-bit
Privilege Mode	U

C.3.2. Format

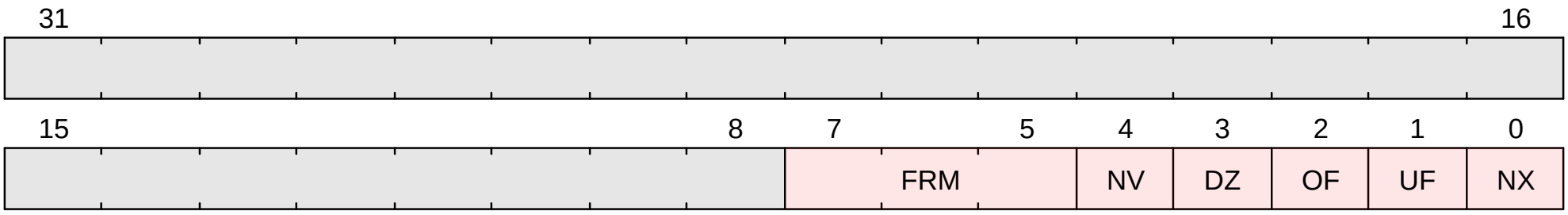


Figure 3. fcsr format

C.3.3. Field Summary

Name	Location	Type	Reset Value
fcsr.FRM	7:5	RW-H	UNDEFINED_LEGAL
fcsr.NV	4	RW-H	UNDEFINED_LEGAL
fcsr.DZ	3	RW-H	UNDEFINED_LEGAL
fcsr.OF	2	RW-H	UNDEFINED_LEGAL
fcsr.UF	1	RW-H	UNDEFINED_LEGAL
fcsr.NX	0	RW-H	UNDEFINED_LEGAL

C.3.4. Fields

fcsr.FRM Field

Location:

7:5

Description:

Rounding modes are encoded as follows:

.Rounding mode encoding.
[%autowidth,float="center",align="center",cols="",<",options="header"]
!===
!Rounding Mode |Mnemonic |Meaning
!000 !RNE !Round to Nearest, ties to Even
!001 !RTZ !Round towards Zero
!010 !RDN !Round Down (towards `-\infty`)
!011 !RUP !Round Up (towards `+\infty`)
!100 !RMM !Round to Nearest, ties to Max Magnitude
!101 ! !*Reserved for future use.*
!110 ! !*Reserved for future use.*
!111 !DYN !In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *reserved*.
!===

A value of 111 in the
instruction's *rm* field selects the dynamic rounding mode held in
frm. The behavior of floating-point instructions that depend on
rounding mode when executed with a reserved rounding mode is *reserved*,
including both static reserved rounding modes (101-110) and dynamic reserved
rounding modes (101-111). Some instructions, including widening conversions,
have the *rm* field but are nevertheless mathematically unaffected by the
rounding mode; software should set their *rm* field to
RNE (000) but implementations must treat the *rm* field as usual (in
particular, with regard to decoding legal vs. reserved encodings).

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.NV Field

Location:

4

Description:

Invalid Operation

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation is invalid and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.DZ Field

Location:

3

Description:

Divide by zero

Cumulative error flag for floating point operations.

Set by hardware when a floating point divide attempts to divide by zero and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.OF Field

Location:

2

Description:

Overflow

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation overflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.UF Field

Location:

1

Description:

Underflow

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation underflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fcsr.NX Field

Location:

0

Description:

Inexact

Cumulative error flag for floating point operations.

Set by hardware when a floating point operation is inexact and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.4. fflags

Floating-Point Accrued Exceptions

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software.

The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

As allowed by the standard, we do not support traps on floating-point exceptions in the F extension, but instead require explicit checks of the flags in software. We considered adding branches controlled directly by the contents of the floating-point accrued exception flags, but ultimately chose to omit these instructions to keep the ISA simple.

C.4.1. Attributes

CSR Address	0x1
Defining extension	F, version >= F@2.2.0
Length	32-bit
Privilege Mode	U

C.4.2. Format

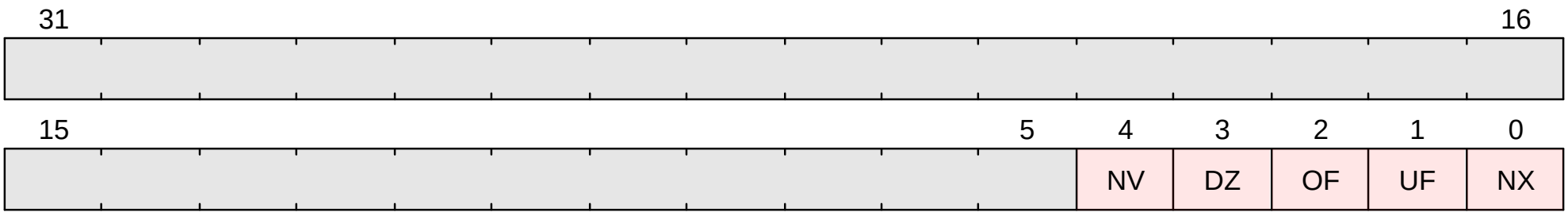


Figure 4. fflags format

C.4.3. Field Summary

Name	Location	Type	Reset Value
fflags.NV	4	RW-H	UNDEFINED_LEGAL
fflags.DZ	3	RW-H	UNDEFINED_LEGAL
fflags.OF	2	RW-H	UNDEFINED_LEGAL
fflags.UF	1	RW-H	UNDEFINED_LEGAL
fflags.NX	0	RW-H	UNDEFINED_LEGAL

C.4.4. Fields

fflags.NV Field

Location:

4

Description:

Set by hardware when a floating point operation is invalid and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.DZ Field

Location:

3

Description:

Set by hardware when a floating point divide attempts to divide by zero and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.OF Field

Location:

2

Description:

Set by hardware when a floating point operation overflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.UF Field

Location:

1

Description:

Set by hardware when a floating point operation underflows and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

fflags.NX Field

Location:

0

Description:

Set by hardware when a floating point operation is inexact and stays set until explicitly cleared by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.4.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
NV = CSR[fcsr].NV = csr_value.NV;
return csr_value.NV;

DZ = CSR[fcsr].DZ = csr_value.DZ;
return csr_value.DZ;

OF = CSR[fcsr].OF = csr_value.OF;
return csr_value.OF;

UF = CSR[fcsr].UF = csr_value.UF;
return csr_value.UF;

NX = CSR[fcsr].NX = csr_value.NX;
return csr_value.NX;
```

C.4.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return (fcsr.NV `<< 4) | (fcsr.DZ `<< 3) | (fcsr.OF `<< 2) | (fcsr.UF `<< 1) | fcsr.NX;
```

C.5. frm

Floating-Point Dynamic Rounding Mode

Rounding modes are encoded as follows:

Table 15. Rounding mode encoding.

!Rounding Mode	Mnemonic	Meaning !000 !RNE !Round to Nearest, ties to Even !001 !RTZ !Round towards Zero !010 !RDN !Round Down (towards `-\infty`) !011 !RUP !Round Up (towards `+\infty`) !100 !RMM !Round to Nearest, ties to Max Magnitude !101 ! !*Reserved for future use.* !110 ! !*Reserved for future use.* !111 !DYN !In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *reserved*.

The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111).

Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

C.5.1. Attributes

CSR Address	0x2
Defining extension	F, version >= F@2.2.0
Length	32-bit
Privilege Mode	U

C.5.2. Format

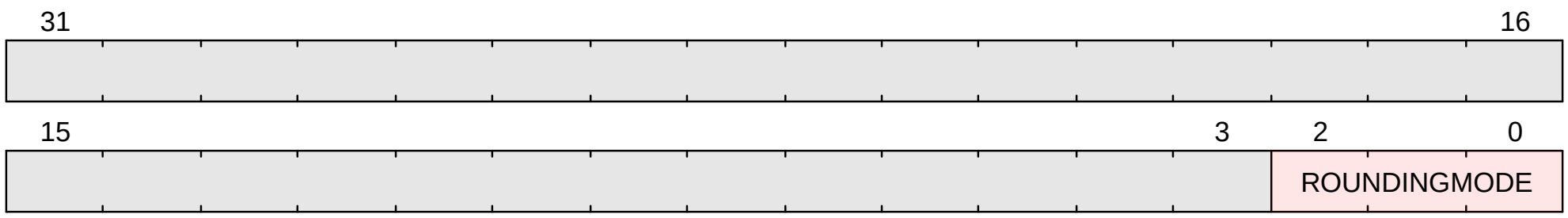


Figure 5. frm format

C.5.3. Field Summary

Name	Location	Type	Reset Value
frm.ROUNDINGMODE	2:0	RW-H	UNDEFINED_LEGAL

C.5.4. Fields

frm.ROUNDINGMODE Field

Location:

2:0

Description:

Rounding mode data.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.5.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
ROUNDINGMODE = CSR[fcsr].FRM = csr_value.ROUNDINGMODE;
return csr_value.ROUNDINGMODE;
```

C.5.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return fcsr.FRM;
```

C.6. hpmcounter10

User-mode Hardware Performance Counter 7

Alias for M-mode CSR mhpmcounter10.

Privilege mode access is controlled with mcounteren.HPM10 <%- if ext?(:S) -%> , scounteren.HPM10 <%- if ext?(:H) -%> , and hcounteren.HPM10 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM10	scounteren.HPM10	hcounteren.HPM10	hpmcounter10 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM10	scounteren.HPM10	hpmcounter10 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM10	hpmcounter10 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.6.1. Attributes

CSR Address	0xc0a
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.6.2. Format

Figure 6. hpmcounter10 format

C.6.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter10.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.6.4. Fields

hpmcounter10.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter10.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.6.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM10 & scounteren.HPM10) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM10 == 1'b0 && mcounteren.HPM10 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM10 & scounteren.HPM10) == 1'b0) && (mcounteren.HPM10 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(10);
```

C.7. hpmcounter11

User-mode Hardware Performance Counter 8

Alias for M-mode CSR mhpmcounter11.

Privilege mode access is controlled with mcounteren.HPM11 <%- if ext?(:S) -%> , scounteren.HPM11 <%- if ext?(:H) -%> , and hcounteren.HPM11 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM11	scounteren.HPM11	hcounteren.HPM11	hpmcounter11 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM11	scounteren.HPM11	hpmcounter11 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM11	hpmcounter11 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.7.1. Attributes

CSR Address	0xc0b
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.7.2. Format

Figure 7. hpmcounter11 format

C.7.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter11.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.7.4. Fields

hpmcounter11.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter11.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.7.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM11 & scounteren.HPM11) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM11 == 1'b0 && mcounteren.HPM11 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM11 & scounteren.HPM11) == 1'b0) && (mcounteren.HPM11 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(11);
```

C.8. hpmcounter12

User-mode Hardware Performance Counter 9

Alias for M-mode CSR mhpmcounter12.

Privilege mode access is controlled with mcounteren.HPM12 <%- if ext?(:S) -%> , scounteren.HPM12 <%- if ext?(:H) -%> , and hcounteren.HPM12 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM12	scounteren.HPM12	hcounteren.HPM12	hpmcounter12 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM12	scounteren.HPM12	hpmcounter12 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM12	hpmcounter12 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.8.1. Attributes

CSR Address	0xc0c
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.8.2. Format

Figure 8. hpmcounter12 format

C.8.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter12.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.8.4. Fields

hpmcounter12.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter12.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.8.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM12 & scounteren.HPM12) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM12 == 1'b0 && mcounteren.HPM12 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM12 & scounteren.HPM12) == 1'b0) && (mcounteren.HPM12 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(12);
```

C.9. hpmcounter13

User-mode Hardware Performance Counter 10

Alias for M-mode CSR mhpmcounter13.

Privilege mode access is controlled with mcounteren.HPM13 <%- if ext?(:S) -%> , scounteren.HPM13 <%- if ext?(:H) -%> , and hcounteren.HPM13 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM13	scounteren.HPM13	hcounteren.HPM13	hpmcounter13 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM13	scounteren.HPM13	hpmcounter13 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM13	hpmcounter13 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.9.1. Attributes

CSR Address	0xc0d
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.9.2. Format

Figure 9. hpmcounter13 format

C.9.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter13.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.9.4. Fields

hpmcounter13.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter13.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.9.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM13 & scounteren.HPM13) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM13 == 1'b0 && mcounteren.HPM13 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM13 & scounteren.HPM13) == 1'b0) && (mcounteren.HPM13 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(13);
```

C.10. hpmcounter14

User-mode Hardware Performance Counter 11

Alias for M-mode CSR mhpmcounter14.

Privilege mode access is controlled with mcounteren.HPM14 <%- if ext?(:S) -%> , scounteren.HPM14 <%- if ext?(:H) -%> , and hcounteren.HPM14 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM14	scounteren.HPM14	hcounteren.HPM14	hpmcounter14 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM14	scounteren.HPM14	hpmcounter14 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM14	hpmcounter14 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.10.1. Attributes

CSR Address	0xc0e
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.10.2. Format

Figure 10. hpmcounter14 format

C.10.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter14.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.10.4. Fields

hpmcounter14.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter14.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.10.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM14 & scounteren.HPM14) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM14 == 1'b0 && mcounteren.HPM14 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM14 & scounteren.HPM14) == 1'b0) && (mcounteren.HPM14 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(14);
```

C.11. hpmcounter15

User-mode Hardware Performance Counter 12

Alias for M-mode CSR mhpmcounter15.

Privilege mode access is controlled with mcounteren.HPM15 <%- if ext?(:S) -%> , scounteren.HPM15 <%- if ext?(:H) -%> , and hcounteren.HPM15 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM15	scounteren.HPM15	hcounteren.HPM15	hpmcounter15 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM15	scounteren.HPM15	hpmcounter15 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM15	hpmcounter15 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.11.1. Attributes

CSR Address	0xc0f
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.11.2. Format

Figure 11. hpmcounter15 format

C.11.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter15.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.11.4. Fields

hpmcounter15.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter15.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.11.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM15 & scounteren.HPM15) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM15 == 1'b0 && mcounteren.HPM15 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM15 & scounteren.HPM15) == 1'b0) && (mcounteren.HPM15 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(15);
```

C.12. hpmcounter16

User-mode Hardware Performance Counter 13

Alias for M-mode CSR mhpmcounter16.

Privilege mode access is controlled with mcounteren.HPM16 <%- if ext?(:S) -%> , scounteren.HPM16 <%- if ext?(:H) -%> , and hcounteren.HPM16 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM16	scounteren.HPM16	hcounteren.HPM16	hpmcounter16 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM16	scounteren.HPM16	hpmcounter16 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM16	hpmcounter16 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.12.1. Attributes

CSR Address	0xc10
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.12.2. Format

Figure 12. hpmcounter16 format

C.12.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter16.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.12.4. Fields

hpmcounter16.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter16.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.12.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM16 & scounteren.HPM16) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM16 == 1'b0 && mcounteren.HPM16 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM16 & scounteren.HPM16) == 1'b0) && (mcounteren.HPM16 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(16);
```

C.13. hpmcounter17

User-mode Hardware Performance Counter 14

Alias for M-mode CSR mhpmcounter17.

Privilege mode access is controlled with mcounteren.HPM17 <%- if ext?(:S) -%> , scounteren.HPM17 <%- if ext?(:H) -%> , and hcounteren.HPM17 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM17	scounteren.HPM17	hcounteren.HPM17	hpmcounter17 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM17	scounteren.HPM17	hpmcounter17 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM17	hpmcounter17 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.13.1. Attributes

CSR Address	0xc11
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.13.2. Format

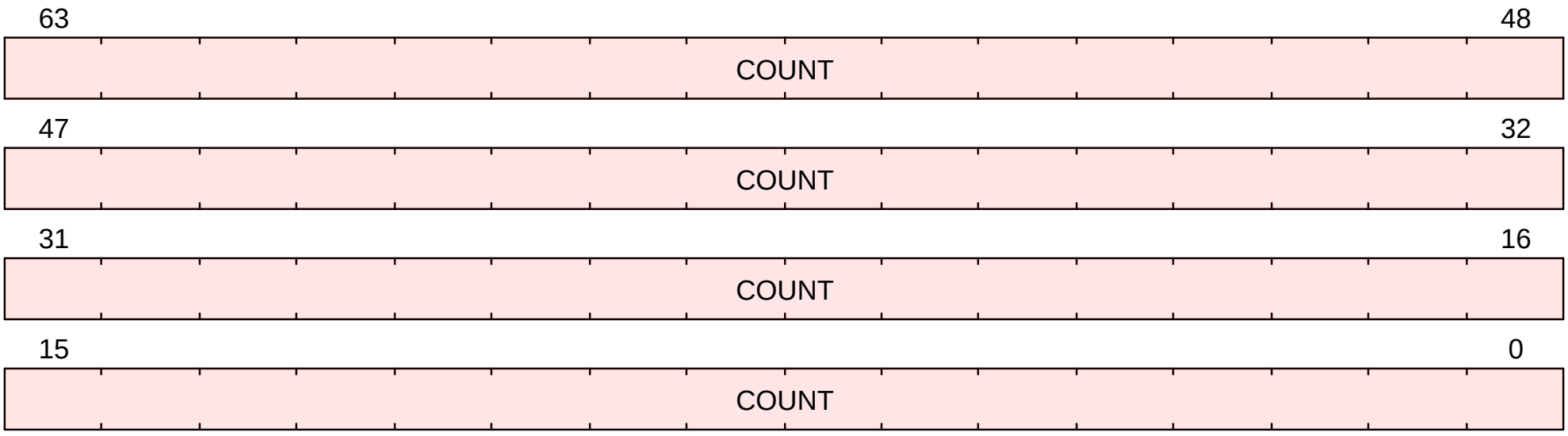


Figure 13. hpmcounter17 format

C.13.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter17.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.13.4. Fields

hpmcounter17.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter17.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.13.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM17 & scounteren.HPM17) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM17 == 1'b0 && mcounteren.HPM17 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM17 & scounteren.HPM17) == 1'b0) && (mcounteren.HPM17 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(17);
```

C.14. hpmcounter18

User-mode Hardware Performance Counter 15

Alias for M-mode CSR mhpmcounter18.

Privilege mode access is controlled with mcounteren.HPM18 <%- if ext?(:S) -%> , scounteren.HPM18 <%- if ext?(:H) -%> , and hcounteren.HPM18 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM18	scounteren.HPM18	hcounteren.HPM18	hpmcounter18 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM18	scounteren.HPM18	hpmcounter18 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM18	hpmcounter18 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.14.1. Attributes

CSR Address	0xc12
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.14.2. Format

Figure 14. hpmcounter18 format

C.14.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter18.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.14.4. Fields

hpmcounter18.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter18.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.14.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM18 & scounteren.HPM18) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM18 == 1'b0 && mcounteren.HPM18 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM18 & scounteren.HPM18) == 1'b0) && (mcounteren.HPM18 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(18);
```

C.15. hpmcounter19

User-mode Hardware Performance Counter 16

Alias for M-mode CSR mhpmcounter19.

Privilege mode access is controlled with mcounteren.HPM19 <%- if ext?(:S) -%> , scounteren.HPM19 <%- if ext?(:H) -%> , and hcounteren.HPM19 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM19	scounteren.HPM19	hcounteren.HPM19	hpmcounter19 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM19	scounteren.HPM19	hpmcounter19 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM19	hpmcounter19 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.15.1. Attributes

CSR Address	0xc13
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.15.2. Format

Figure 15. hpmcounter19 format

C.15.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter19.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.15.4. Fields

hpmcounter19.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter19.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.15.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM19 & scounteren.HPM19) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM19 == 1'b0 && mcounteren.HPM19 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM19 & scounteren.HPM19) == 1'b0) && (mcounteren.HPM19 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(19);
```

C.16. hpmcounter20

User-mode Hardware Performance Counter 17

Alias for M-mode CSR mhpmcounter20.

Privilege mode access is controlled with mcounteren.HPM20 <%- if ext?(:S) -%> , scounteren.HPM20 <%- if ext?(:H) -%> , and hcounteren.HPM20 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM20	scounteren.HPM20	hcounteren.HPM20	hpmcounter20 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM20	scounteren.HPM20	hpmcounter20 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM20	hpmcounter20 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.16.1. Attributes

CSR Address	0xc14
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.16.2. Format

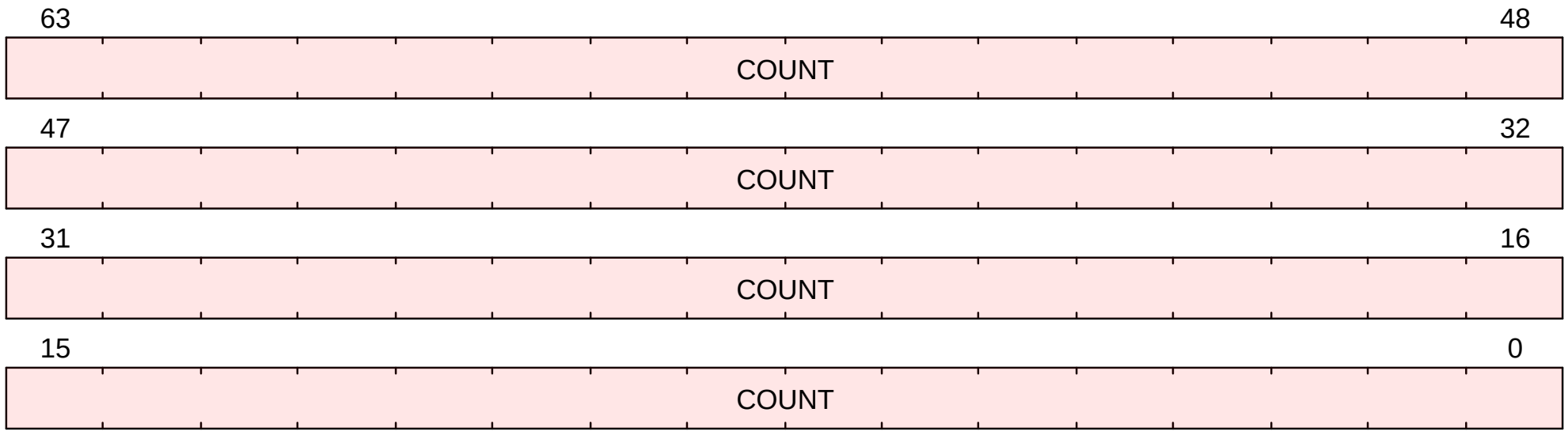


Figure 16. hpmcounter20 format

C.16.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter20.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.16.4. Fields

hpmcounter20.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter20.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.16.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM20 & scounteren.HPM20) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM20 == 1'b0 && mcounteren.HPM20 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM20 & scounteren.HPM20) == 1'b0) && (mcounteren.HPM20 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(20);
```

C.17. hpmcounter21

User-mode Hardware Performance Counter 18

Alias for M-mode CSR mhpmcounter21.

Privilege mode access is controlled with mcounteren.HPM21 <%- if ext?(:S) -%> , scounteren.HPM21 <%- if ext?(:H) -%> , and hcounteren.HPM21 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM21	scounteren.HPM21	hcounteren.HPM21	hpmcounter21 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM21	scounteren.HPM21	hpmcounter21 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM21	hpmcounter21 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.17.1. Attributes

CSR Address	0xc15
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.17.2. Format

Figure 17. hpmcounter21 format

C.17.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter21.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.17.4. Fields

hpmcounter21.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter21.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.17.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM21 & scounteren.HPM21) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM21 == 1'b0 && mcounteren.HPM21 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM21 & scounteren.HPM21) == 1'b0) && (mcounteren.HPM21 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(21);
```

C.18. hpmcounter22

User-mode Hardware Performance Counter 19

Alias for M-mode CSR mhpmcounter22.

Privilege mode access is controlled with mcounteren.HPM22 <%- if ext?(:S) -%> , scounteren.HPM22 <%- if ext?(:H) -%> , and hcounteren.HPM22 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM22	scounteren.HPM22	hcounteren.HPM22	hpmcounter22 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM22	scounteren.HPM22	hpmcounter22 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM22	hpmcounter22 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.18.1. Attributes

CSR Address	0xc16
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.18.2. Format

Figure 18. hpmcounter22 format

C.18.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter22.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.18.4. Fields

hpmcounter22.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter22.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.18.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM22 & scounteren.HPM22) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM22 == 1'b0 && mcounteren.HPM22 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM22 & scounteren.HPM22) == 1'b0) && (mcounteren.HPM22 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(22);
```

C.19. hpmcounter23

User-mode Hardware Performance Counter 20

Alias for M-mode CSR mhpmcounter23.

Privilege mode access is controlled with mcounteren.HPM23 <%- if ext?(:S) -%> , scounteren.HPM23 <%- if ext?(:H) -%> , and hcounteren.HPM23 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM23	scounteren.HPM23	hcounteren.HPM23	hpmcounter23 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM23	scounteren.HPM23	hpmcounter23 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM23	hpmcounter23 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.19.1. Attributes

CSR Address	0xc17
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.19.2. Format

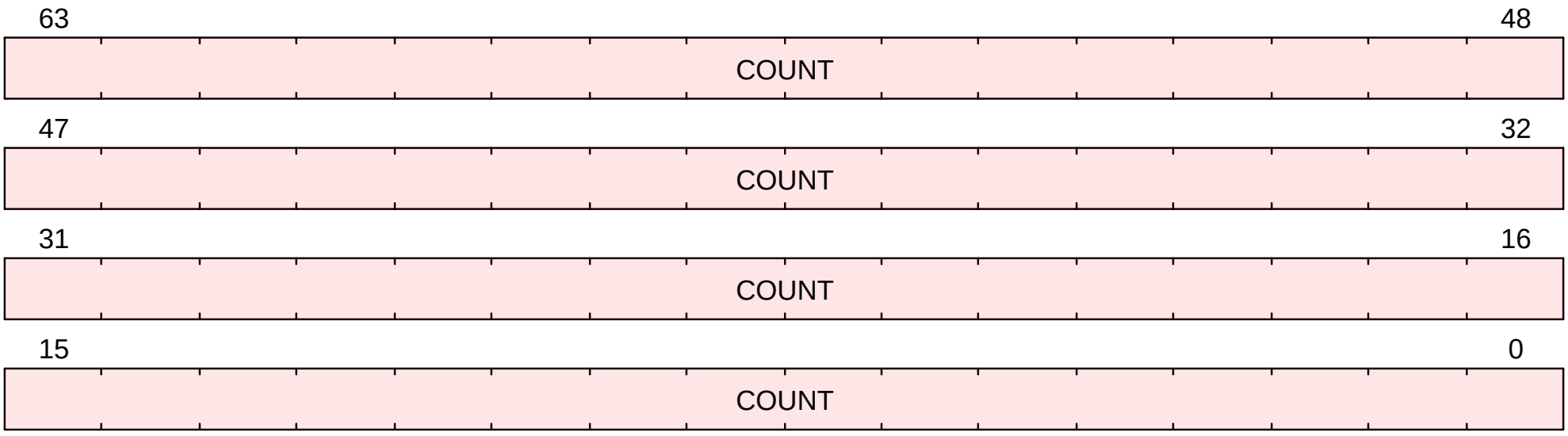


Figure 19. hpmcounter23 format

C.19.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter23.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.19.4. Fields

hpmcounter23.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter23.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.19.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM23 & scounteren.HPM23) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM23 == 1'b0 && mcounteren.HPM23 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM23 & scounteren.HPM23) == 1'b0) && (mcounteren.HPM23 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(23);
```

C.20. hpmcounter24

User-mode Hardware Performance Counter 21

Alias for M-mode CSR mhpmcounter24.

Privilege mode access is controlled with mcounteren.HPM24 <%- if ext?(:S) -%> , scounteren.HPM24 <%- if ext?(:H) -%> , and hcounteren.HPM24 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM24	scounteren.HPM24	hcounteren.HPM24	hpmcounter24 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM24	scounteren.HPM24	hpmcounter24 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM24	hpmcounter24 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.20.1. Attributes

CSR Address	0xc18
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.20.2. Format

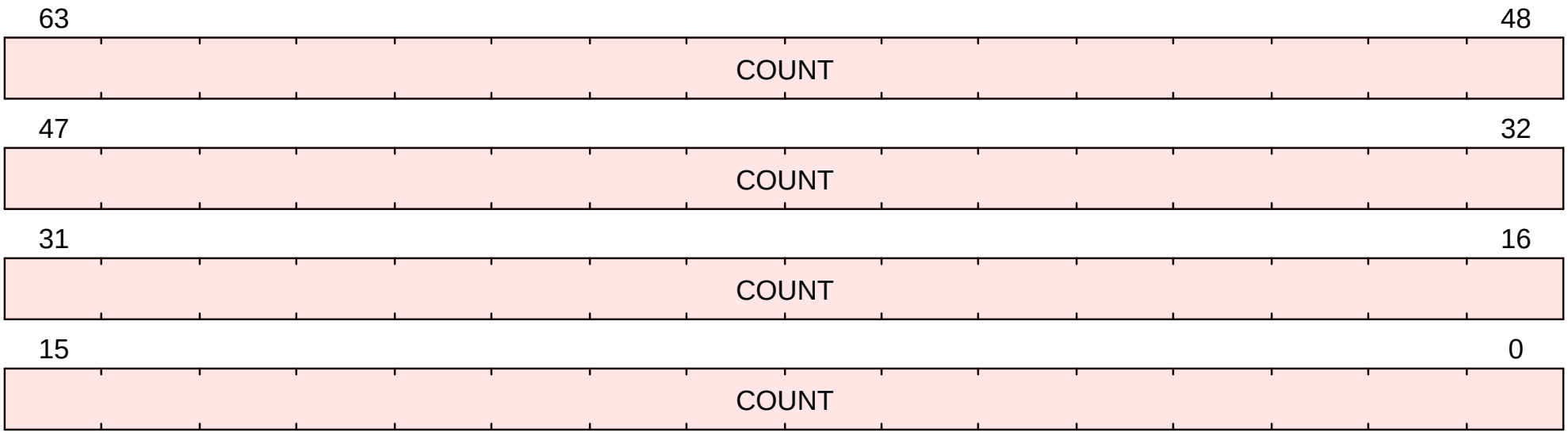


Figure 20. hpmcounter24 format

C.20.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter24.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.20.4. Fields

hpmcounter24.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter24.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.20.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM24 & scounteren.HPM24) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM24 == 1'b0 && mcounteren.HPM24 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM24 & scounteren.HPM24) == 1'b0) && (mcounteren.HPM24 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(24);
```

C.21. hpmcounter25

User-mode Hardware Performance Counter 22

Alias for M-mode CSR mhpmcounter25.

Privilege mode access is controlled with mcounteren.HPM25 <%- if ext?(:S) -%> , scounteren.HPM25 <%- if ext?(:H) -%> , and hcounteren.HPM25 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM25	scounteren.HPM25	hcounteren.HPM25	hpmcounter25 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM25	scounteren.HPM25	hpmcounter25 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM25	hpmcounter25 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.21.1. Attributes

CSR Address	0xc19
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.21.2. Format

Figure 21. hpmcounter25 format

C.21.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter25.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.21.4. Fields

hpmcounter25.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter25.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.21.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM25 & scounteren.HPM25) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM25 == 1'b0 && mcounteren.HPM25 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM25 & scounteren.HPM25) == 1'b0) && (mcounteren.HPM25 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(25);
```

C.22. hpmcounter26

User-mode Hardware Performance Counter 23

Alias for M-mode CSR mhpmcounter26.

Privilege mode access is controlled with mcounteren.HPM26 <%- if ext?(:S) -%> , scounteren.HPM26 <%- if ext?(:H) -%> , and hcounteren.HPM26 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM26	scounteren.HPM26	hcounteren.HPM26	hpmcounter26 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM26	scounteren.HPM26	hpmcounter26 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM26	hpmcounter26 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.22.1. Attributes

CSR Address	0xc1a
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.22.2. Format

Figure 22. hpmcounter26 format

C.22.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter26.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.22.4. Fields

hpmcounter26.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter26.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.22.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM26 & scounteren.HPM26) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM26 == 1'b0 && mcounteren.HPM26 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM26 & scounteren.HPM26) == 1'b0) && (mcounteren.HPM26 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(26);
```

C.23. hpmcounter27

User-mode Hardware Performance Counter 24

Alias for M-mode CSR mhpmcounter27.

Privilege mode access is controlled with mcounteren.HPM27 <%- if ext?(:S) -%> , scounteren.HPM27 <%- if ext?(:H) -%> , and hcounteren.HPM27 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM27	scounteren.HPM27	hcounteren.HPM27	hpmcounter27 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM27	scounteren.HPM27	hpmcounter27 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM27	hpmcounter27 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.23.1. Attributes

CSR Address	0xc1b
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.23.2. Format

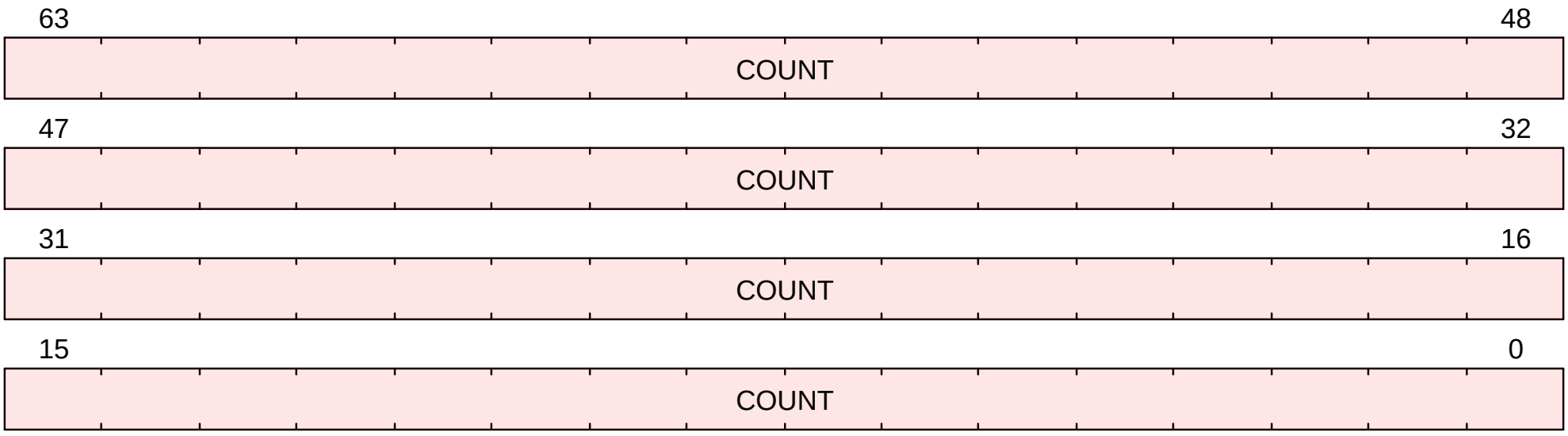


Figure 23. hpmcounter27 format

C.23.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter27.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.23.4. Fields

hpmcounter27.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter27.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.23.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM27 & scounteren.HPM27) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM27 == 1'b0 && mcounteren.HPM27 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM27 & scounteren.HPM27) == 1'b0) && (mcounteren.HPM27 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(27);
```

C.24. hpmcounter28

User-mode Hardware Performance Counter 25

Alias for M-mode CSR mhpmcounter28.

Privilege mode access is controlled with mcounteren.HPM28 <%- if ext?(:S) -%> , scounteren.HPM28 <%- if ext?(:H) -%> , and hcounteren.HPM28 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM28	scounteren.HPM28	hcounteren.HPM28	hpmcounter28 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM28	scounteren.HPM28	hpmcounter28 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM28	hpmcounter28 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.24.1. Attributes

CSR Address	0xc1c
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.24.2. Format

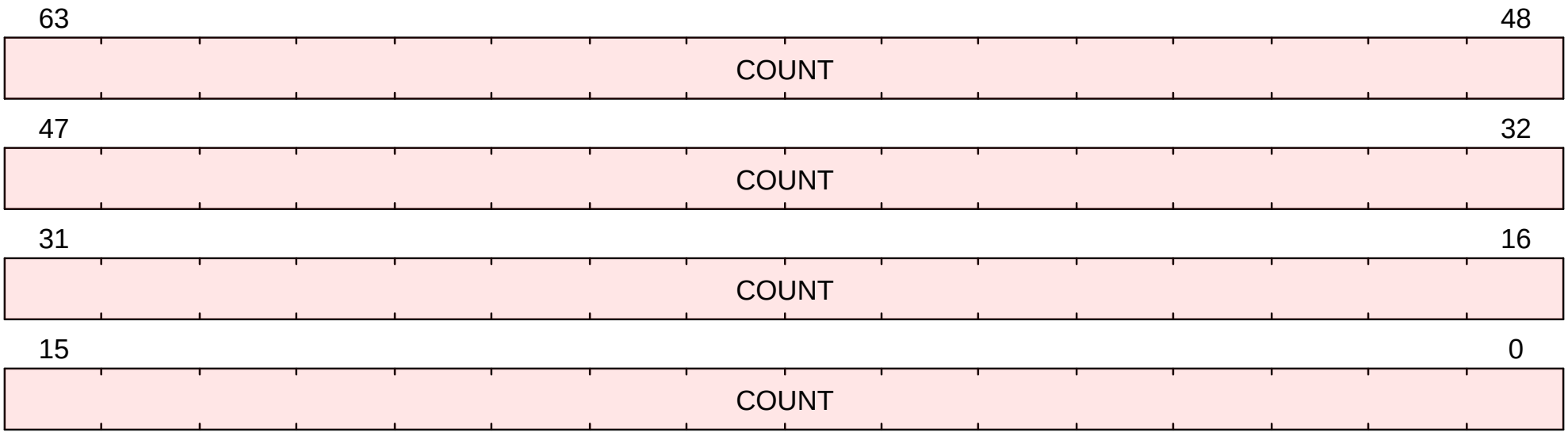


Figure 24. hpmcounter28 format

C.24.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter28.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.24.4. Fields

hpmcounter28.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter28.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.24.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM28 & scounteren.HPM28) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM28 == 1'b0 && mcounteren.HPM28 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM28 & scounteren.HPM28) == 1'b0) && (mcounteren.HPM28 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(28);
```

C.25. hpmcounter29

User-mode Hardware Performance Counter 26

Alias for M-mode CSR mhpmcounter29.

Privilege mode access is controlled with mcounteren.HPM29 <%- if ext?(:S) -%> , scounteren.HPM29 <%- if ext?(:H) -%> , and hcounteren.HPM29 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM29	scounteren.HPM29	hcounteren.HPM29	hpmcounter29 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM29	scounteren.HPM29	hpmcounter29 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM29	hpmcounter29 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.25.1. Attributes

CSR Address	0xc1d
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.25.2. Format

Figure 25. hpmcounter29 format

C.25.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter29.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.25.4. Fields

hpmcounter29.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter29.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.25.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM29 & scounteren.HPM29) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM29 == 1'b0 && mcounteren.HPM29 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM29 & scounteren.HPM29) == 1'b0) && (mcounteren.HPM29 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(29);
```

C.26. hpmcounter3

User-mode Hardware Performance Counter 0

Alias for M-mode CSR mhpmcounter3.

Privilege mode access is controlled with mcounteren.HPM3 <%- if ext?(:S) -%> , scounteren.HPM3 <%- if ext?(:H) -%> , and hcounteren.HPM3 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM3	scounteren.HPM3	hcounteren.HPM3	hpmcounter3 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM3	scounteren.HPM3	hpmcounter3 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM3	hpmcounter3 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.26.1. Attributes

CSR Address	0xc03
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.26.2. Format

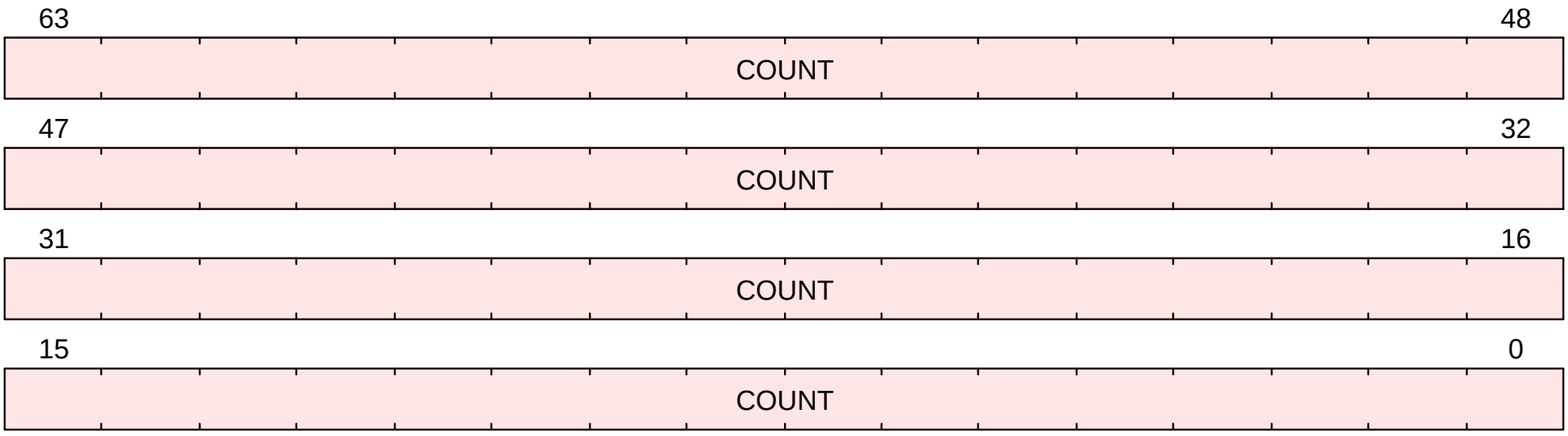


Figure 26. hpmcounter3 format

C.26.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter3.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.26.4. Fields

hpmcounter3.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter3.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.26.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM3 & scounteren.HPM3) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM3 == 1'b0 && mcounteren.HPM3 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM3 & scounteren.HPM3) == 1'b0) && (mcounteren.HPM3 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(3);
```

C.27. hpmcounter30

User-mode Hardware Performance Counter 27

Alias for M-mode CSR mhpmcounter30.

Privilege mode access is controlled with mcounteren.HPM30 <%- if ext?(:S) -%> , scounteren.HPM30 <%- if ext?(:H) -%> , and hcounteren.HPM30 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM30	scounteren.HPM30	hcounteren.HPM30	hpmcounter30 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM30	scounteren.HPM30	hpmcounter30 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM30	hpmcounter30 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.27.1. Attributes

CSR Address	0xc1e
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.27.2. Format

Figure 27. hpmcounter30 format

C.27.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter30.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.27.4. Fields

hpmcounter30.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter30.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.27.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM30 & scounteren.HPM30) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM30 == 1'b0 && mcounteren.HPM30 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM30 & scounteren.HPM30) == 1'b0) && (mcounteren.HPM30 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(30);
```

C.28. hpmcounter31

User-mode Hardware Performance Counter 28

Alias for M-mode CSR mhpmcounter31.

Privilege mode access is controlled with mcounteren.HPM31 <%- if ext?(:S) -%> , scounteren.HPM31 <%- if ext?(:H) -%> , and hcounteren.HPM31 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM31	scounteren.HPM31	hcounteren.HPM31	hpmcounter31 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM31	scounteren.HPM31	hpmcounter31 behavior	
		S-mode	U-mode
0	-	IllegalInstruction	IllegalInstruction
1	0	read-only	IllegalInstruction
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM31	hpmcounter31 behavior
	U-mode
0	IllegalInstruction
1	read-only

<%- end -%>

C.28.1. Attributes

CSR Address	0xc1f
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.28.2. Format

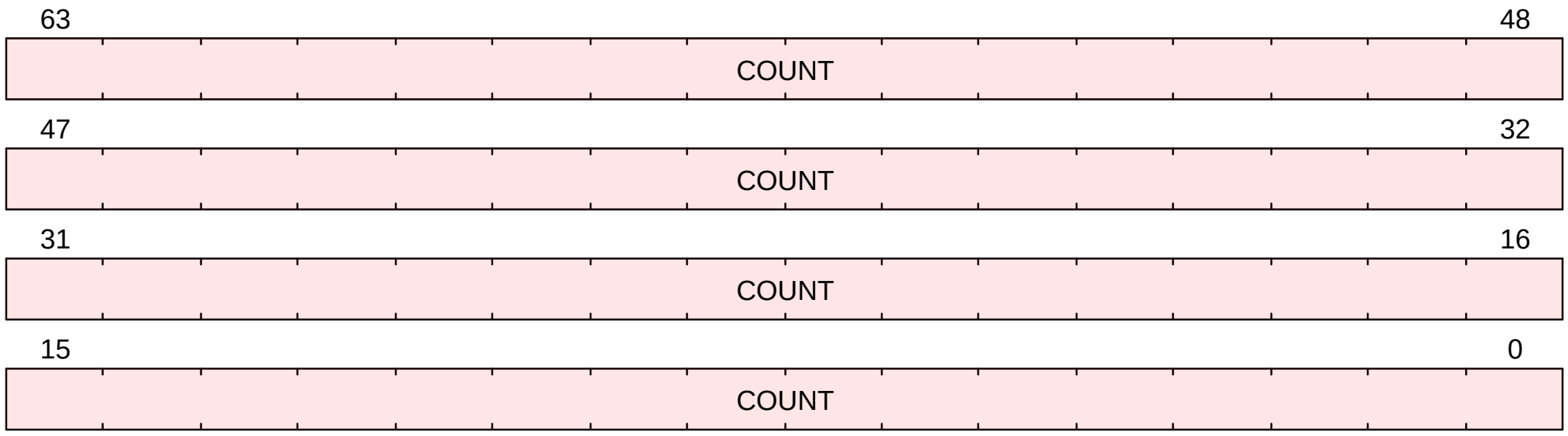


Figure 28. hpmcounter31 format

C.28.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter31.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.28.4. Fields

hpmcounter31.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter31.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.28.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM31 & scounteren.HPM31) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM31 == 1'b0 && mcounteren.HPM31 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM31 & scounteren.HPM31) == 1'b0) && (mcounteren.HPM31 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(31);
```

C.29. hpmcounter4

User-mode Hardware Performance Counter 1

Alias for M-mode CSR mhpmcounter4.

Privilege mode access is controlled with mcounteren.HPM4 <%- if ext?(:S) -%> , scounteren.HPM4 <%- if ext?(:H) -%> , and hcounteren.HPM4 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM4	scounteren.HPM4	hcounteren.HPM4	hpmcounter4 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM4	scounteren.HPM4	hpmcounter4 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM4	hpmcounter4 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.29.1. Attributes

CSR Address	0xc04
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.29.2. Format

Figure 29. hpmcounter4 format

C.29.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter4.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.29.4. Fields

hpmcounter4.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter4.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.29.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM4 & scounteren.HPM4) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM4 == 1'b0 && mcounteren.HPM4 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM4 & scounteren.HPM4) == 1'b0) && (mcounteren.HPM4 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(4);
```

C.30. hpmcounter5

User-mode Hardware Performance Counter 2

Alias for M-mode CSR mhpmcounter5.

Privilege mode access is controlled with mcounteren.HPM5 <%- if ext?(:S) -%> , scounteren.HPM5 <%- if ext?(:H) -%> , and hcounteren.HPM5 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM5	scounteren.HPM5	hcounteren.HPM5	hpmcounter5 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM5	scounteren.HPM5	hpmcounter5 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM5	hpmcounter5 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.30.1. Attributes

CSR Address	0xc05
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.30.2. Format

Figure 30. hpmcounter5 format

C.30.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter5.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.30.4. Fields

hpmcounter5.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter5.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.30.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM5 & scounteren.HPM5) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM5 == 1'b0 && mcounteren.HPM5 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM5 & scounteren.HPM5) == 1'b0) && (mcounteren.HPM5 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(5);
```

C.31. hpmcounter6

User-mode Hardware Performance Counter 3

Alias for M-mode CSR mhpmcounter6.

Privilege mode access is controlled with mcounteren.HPM6 <%- if ext?(:S) -%> , scounteren.HPM6 <%- if ext?(:H) -%> , and hcounteren.HPM6 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM6	scounteren.HPM6	hcounteren.HPM6	hpmcounter6 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM6	scounteren.HPM6	hpmcounter6 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM6	hpmcounter6 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.31.1. Attributes

CSR Address	0xc06
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.31.2. Format

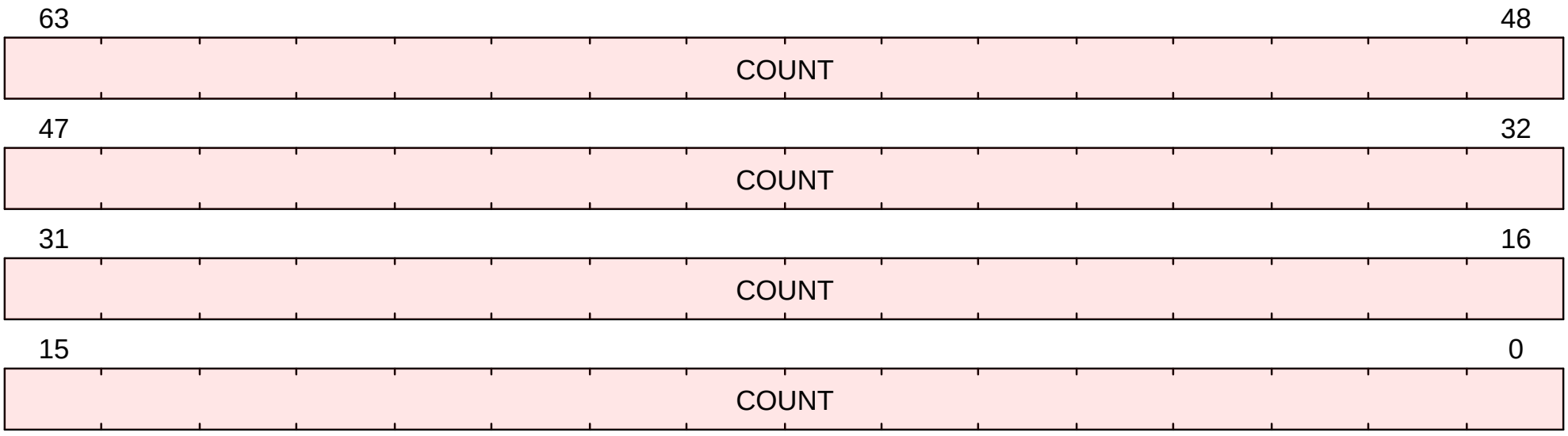


Figure 31. hpmcounter6 format

C.31.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter6.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.31.4. Fields

hpmcounter6.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter6.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.31.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM6 & scounteren.HPM6) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM6 == 1'b0 && mcounteren.HPM6 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM6 & scounteren.HPM6) == 1'b0) && (mcounteren.HPM6 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(6);
```

C.32. hpmcounter7

User-mode Hardware Performance Counter 4

Alias for M-mode CSR mhpmcounter7.

Privilege mode access is controlled with mcounteren.HPM7 <%- if ext?(:S) -%> , scounteren.HPM7 <%- if ext?(:H) -%> , and hcounteren.HPM7 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM7	scounteren.HPM7	hcounteren.HPM7	hpmcounter7 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM7	scounteren.HPM7	hpmcounter7 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM7	hpmcounter7 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.32.1. Attributes

CSR Address	0xc07
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.32.2. Format

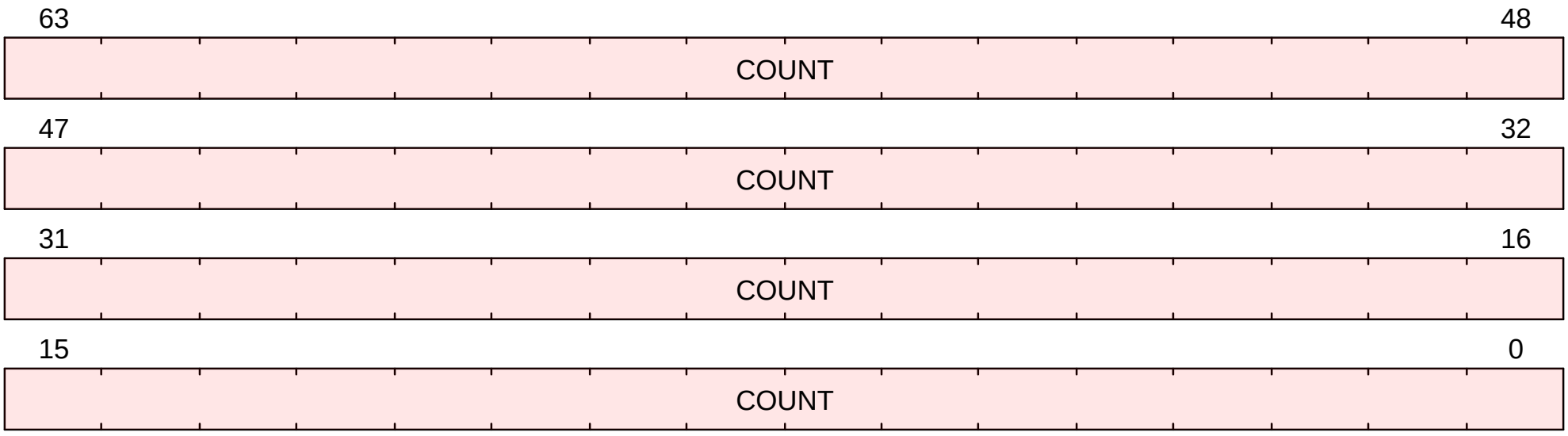


Figure 32. hpmcounter7 format

C.32.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter7.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.32.4. Fields

hpmcounter7.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter7.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.32.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM7 & scounteren.HPM7) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM7 == 1'b0 && mcounteren.HPM7 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM7 & scounteren.HPM7) == 1'b0) && (mcounteren.HPM7 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(7);
```

C.33. hpmcounter8

User-mode Hardware Performance Counter 5

Alias for M-mode CSR mhpmcounter8.

Privilege mode access is controlled with mcounteren.HPM8 <%- if ext?(:S) -%> , scounteren.HPM8 <%- if ext?(:H) -%> , and hcounteren.HPM8 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM8	scounteren.HPM8	hcounteren.HPM8	hpmcounter8 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM8	scounteren.HPM8	hpmcounter8 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM8	hpmcounter8 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.33.1. Attributes

CSR Address	0xc08
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.33.2. Format

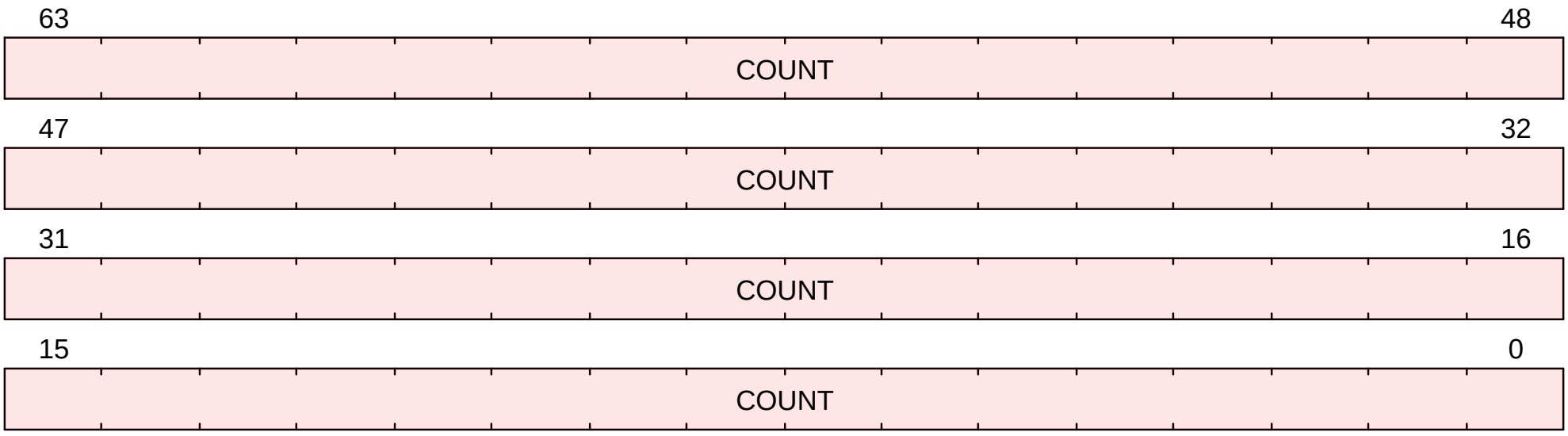


Figure 33. hpmcounter8 format

C.33.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter8.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.33.4. Fields

hpmcounter8.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter8.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.33.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM8 & scounteren.HPM8) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM8 == 1'b0 && mcounteren.HPM8 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM8 & scounteren.HPM8) == 1'b0) && (mcounteren.HPM8 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(8);
```

C.34. hpmcounter9

User-mode Hardware Performance Counter 6

Alias for M-mode CSR mhpmcounter9.

Privilege mode access is controlled with mcounteren.HPM9 <%- if ext?(:S) -%> , scounteren.HPM9 <%- if ext?(:H) -%> , and hcounteren.HPM9 <%- end -%> <%- end -%> as follows:

<%- if ext?(:H) -%>

mcounteren.HPM9	scounteren.HPM9	hcounteren.HPM9	hpmcounter9 behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`	`IllegalInstruction`
1	0	0	read-only	`IllegalInstruction`	`VirtualInstruction`	`VirtualInstruction`
1	1	0	read-only	read-only	`VirtualInstruction`	`VirtualInstruction`
1	0	1	read-only	`IllegalInstruction`	read-only	`VirtualInstruction`
1	1	1	read-only	read-only	read-only	read-only

<%- elsif ext?(:S) -%>

mcounteren.HPM9	scounteren.HPM9	hpmcounter9 behavior	
		S-mode	U-mode
0	-	`IllegalInstruction`	`IllegalInstruction`
1	0	read-only	`IllegalInstruction`
1	1	read-only	read-only

<%- else -%>

mcounteren.HPM9	hpmcounter9 behavior
	U-mode
0	`IllegalInstruction`
1	read-only

<%- end -%>

C.34.1. Attributes

CSR Address	0xc09
Defining extension	Zihpm, version >= Zihpm@2.0.0
Length	64-bit
Privilege Mode	U

C.34.2. Format

Figure 34. hpmcounter9 format

C.34.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter9.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.34.4. Fields

hpmcounter9.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter9.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.34.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM9 & scounteren.HPM9) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM9 == 1'b0 && mcounteren.HPM9 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM9 & scounteren.HPM9) == 1'b0) && (mcounteren.HPM9 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(9);
```

C.35. instret

Instructions retired counter for RDINSTRET Instruction

Alias for M-mode CSR minstret.

Privilege mode access is controlled with mcounteren.IR, scounteren.IR, and hcounteren.IR as follows:

mcounteren.IR	scounteren.IR	hcounteren.IR	instret behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.35.1. Attributes

CSR Address	0xc02
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

C.35.2. Format

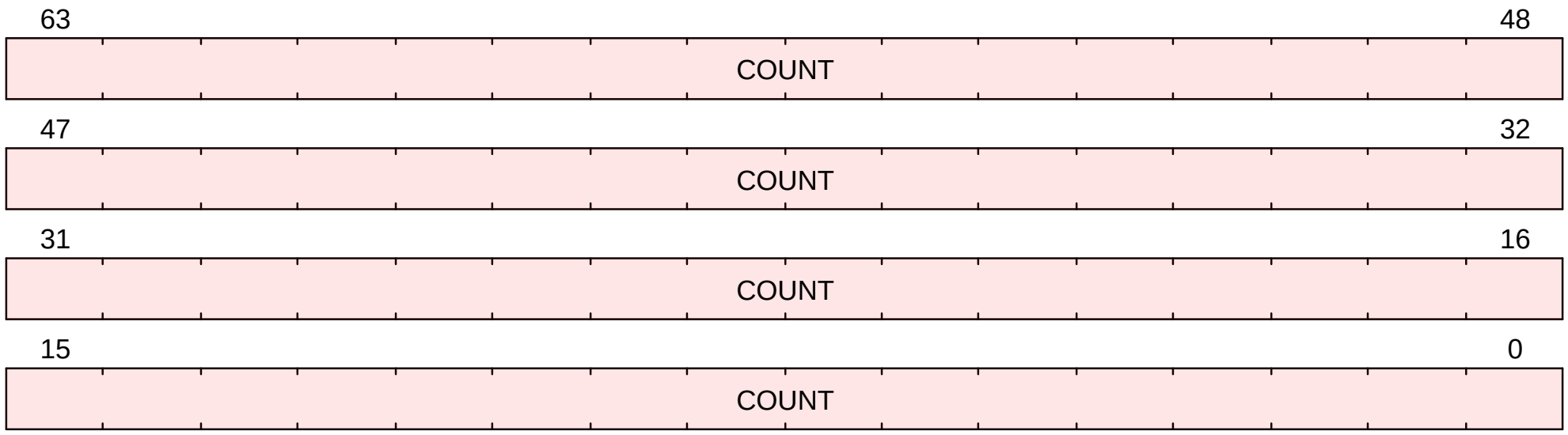


Figure 35. instret format

C.35.3. Field Summary

Name	Location	Type	Reset Value
instret.COUNT	63:0	RO-H	0

C.35.4. Fields

instret.COUNT Field

Location:

63:0

Description:

Alias of minstret.COUNT.

Type:

RO-H

Reset value:

0

C.35.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.IR & scounteren.IR) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.IR == 1'b0 && mcounteren.IR == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.IR & scounteren.IR) == 1'b0) && (mcounteren.IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return minstret.COUNT;
```

C.36. instreth

Instructions retired counter, high bits

instreth is only defined in RV32.

Alias for high bits of M-mode CSR minstret[63:32].

Privilege mode access is controlled with mcounteren.IR, scounteren.IR, and hcounteren.IR as follows:

mcounteren.IR	scounteren.IR	hcounteren.IR	instret behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.36.1. Attributes

CSR Address	0xc82
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

C.36.2. Format

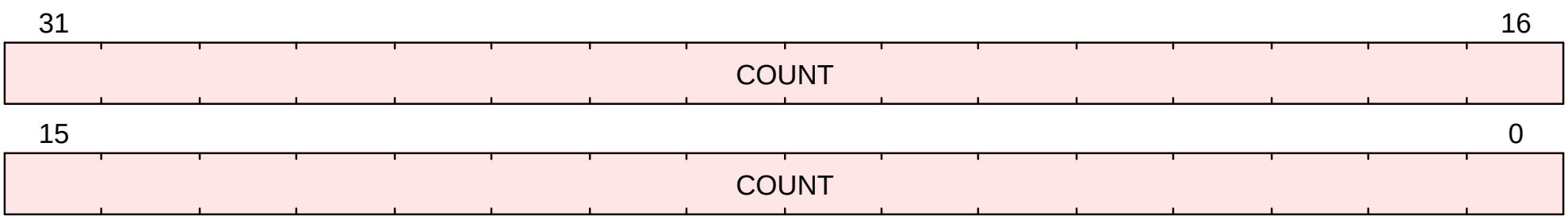


Figure 36. instreth format

C.36.3. Field Summary

Name	Location	Type	Reset Value
instreth.COUNT	31:0	RO-H	UNDEFINED_LEGAL

C.36.4. Fields

instreth.COUNT Field

Location:

31:0

Description:

Alias of minstret.COUNT[63:32].

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.36.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.IR & scounteren.IR) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.IR == 1'b0 && mcounteren.IR == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.IR & scounteren.IR) == 1'b0) && (mcounteren.IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle();
```

C.37. mcycle

Machine Cycle Counter

Counts the number of clock cycles executed by the processor core on which the hart is running. The counter has 64-bit precision on all RV32 and RV64 harts.

The mcycle CSR may be shared between harts on the same core, in which case writes to mcycle will be visible to those harts. The platform should provide a mechanism to indicate which harts share an mcycle CSR.

C.37.1. Attributes

CSR Address	0xb00
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	M

C.37.2. Format

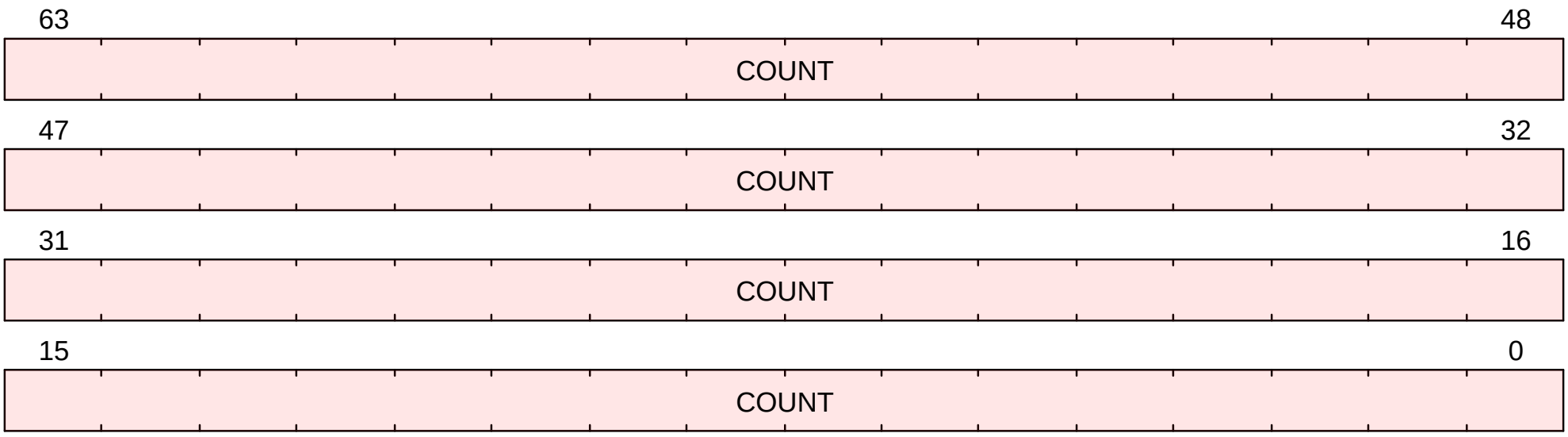


Figure 37. mcycle format

C.37.3. Field Summary

Name	Location	Type	Reset Value
mcycle.COUNT	63:0	RW-RH	UNDEFINED_LEGAL

C.37.4. Fields

mcycle.COUNT Field

Location:

63:0

Description:

Cycle counter.

<%- if ext?(:Zicntr) -%>
Aliased as cycle.
<%- end -%>

Increments every cycle unless:

- mcountinhibit.CY <%- if ext?(:Smcdeleg) -%>or its alias `scountinhibit.CY`<%- end -%> is set
  <%- if ext?(:Smcntrpmf) -%>
- mcyclecfg.MINH is set and the current privilege level is M
  <%- if ext?(:S) -%>
- mcyclecfg.SINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is (H)S
  <%- end -%>

<%- if ext?(:U) -%>

- mcyclecfg.UINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is U
  <%- end -%>
  <%- if ext?(:H) -%>
- mcyclecfg.VSINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is VS
- mcyclecfg.VUINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is VU
  <%- end -%>
  <%- end -%>

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

C.37.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
  return sw_write_mcycle({read_mcycle()[63:31], csr_value.COUNT[31:0]});
} else {
  return sw_write_mcycle(csr_value.COUNT);
}
```

C.37.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return read_mcycle();
```

C.38. mcycleh

High-half machine Cycle Counter

mcycleh is only defined in RV32.

High-half alias of mcycle.

C.38.1. Attributes

CSR Address	0xb80
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	M

C.38.2. Format

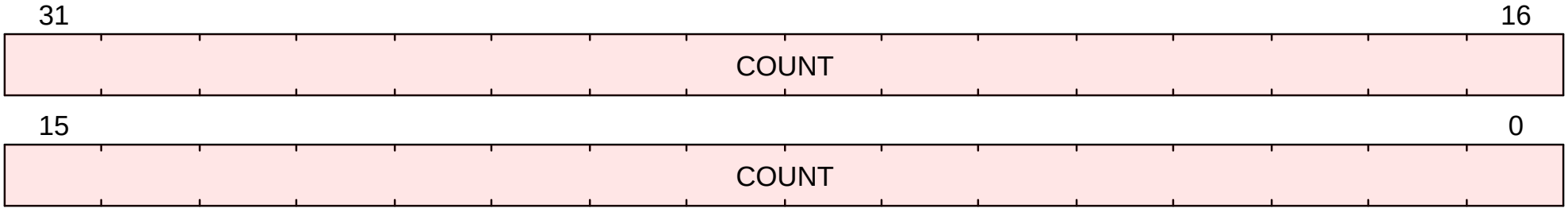


Figure 38. mcycleh format

C.38.3. Field Summary

Name	Location	Type	Reset Value
mcycleh.COUNT	31:0	RW-RH	UNDEFINED_LEGAL

C.38.4. Fields

mcycleh.COUNT Field

Location:

31:0

Description:

Alias of upper half of mcycle.COUNT.

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

C.38.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
  return sw_write_mcycle({csr_value.COUNT[31:0], read_mcycle()[31:0]});
} else {
  return sw_write_mcycle(csr_value.COUNT);
}
```

C.38.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return read_mcycle()[63:32];
```

C.39. minstret

Machine Instructions Retired Counter

Counts the number of instructions retired by this hart from some arbitrary start point in the past.

Instructions that cause synchronous exceptions, including ecall and ebreak, are not considered to retire and hence do not increment the minstret CSR.

C.39.1. Attributes

CSR Address	0xb02
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	M

C.39.2. Format

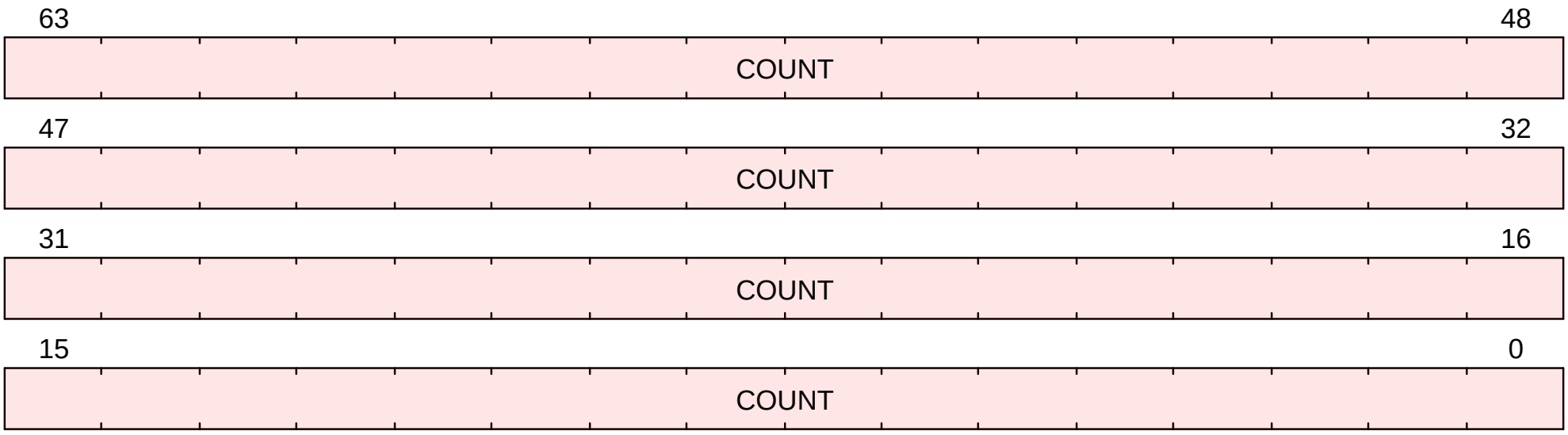


Figure 39. minstret format

C.39.3. Field Summary

Name	Location	Type	Reset Value
minstret.COUNT	63:0	RW-H	UNDEFINED_LEGAL

C.39.4. Fields

minstret.COUNT Field

Location:

63:0

Description:

Instructions retired counter.

<%- if ext?(:Zicntr) -%>
Aliased as instret.COUNT.
<%- end -%>

Increments every time an instruction retires unless:

- mcountinhibit.IR <%- if ext?(:Smcdeleg) -%>or its alias `scountinhibit.IR`<%- end -%> is set
  <%- if ext?(:Smcntrpmf) -%>
- minstretcfg.MINH is set and the current privilege level is M
  <%- if ext?(:S) -%>
- minstretcfg.SINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is (H)S
  <%- end -%>
  <%- if ext?(:U) -%>
- minstretcfg.UINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is U

<%- end -%>
<%- if ext?(:H) -%>

- minstretcfg.VSINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is VS
- minstretcfg.VUINH <%- if ext?(:Ssccfg) -%>or its alias `instretcfg.SINH`<%- end -%> is set and the current privilege level is VU
<%- end -%>
<%- end -%>

An instruction that causes an exception, notably including MRET/SRET,
does not retire and does not cause minstret.COUNT to increment.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.40. minstreth

Machine Instructions Retired Counter

minstreth is only defined in RV32.

Upper half of 64-bit instructions retired counters.

See minstret for details.

C.40.1. Attributes

CSR Address	0xb82
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	M

C.40.2. Format

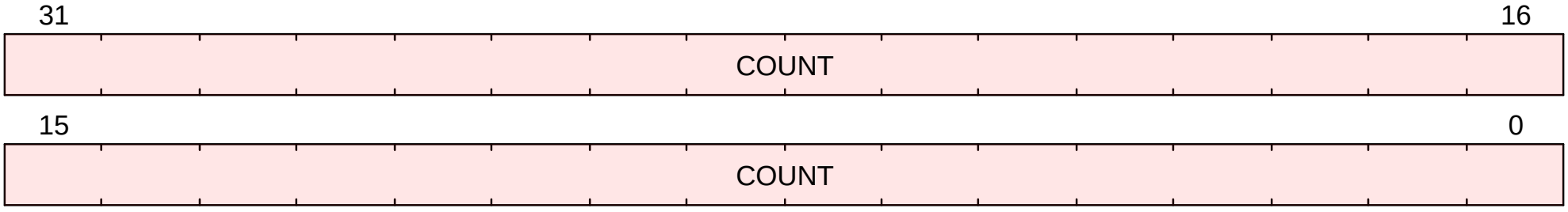


Figure 40. minstreth format

C.40.3. Field Summary

Name	Location	Type	Reset Value
minstreth.COUNT	31:0	RW-H	UNDEFINED_LEGAL

C.40.4. Fields

minstreth.COUNT Field

Location:

31:0

Description:

Instructions retired counter

Upper half of minstret.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.40.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = CSR[mcycle].COUNT = {csr_value.COUNT[31:0], CSR[minstret].COUNT[31:0]};
return csr_value.COUNT;
```

C.40.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return minstret.COUNT[63:32];
```

C.41. time

Timer for RDTIME Instruction

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR `mtime`.

Privilege mode access is controlled with mcounteren.TM, scounteren.TM, and hcounteren.TM as follows:

mcounteren.TM	scounteren.TM	scounteren.TM	time behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	0	0	read-only	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	1	0	read-only	read-only	Illegal Instruction	Illegal Instruction
1	0	1	read-only	Illegal Instruction	read-only	Illegal Instruction
1	1	1	read-only	read-only	read-only	read-only

C.41.1. Attributes

CSR Address	0xc01
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

C.41.2. Format

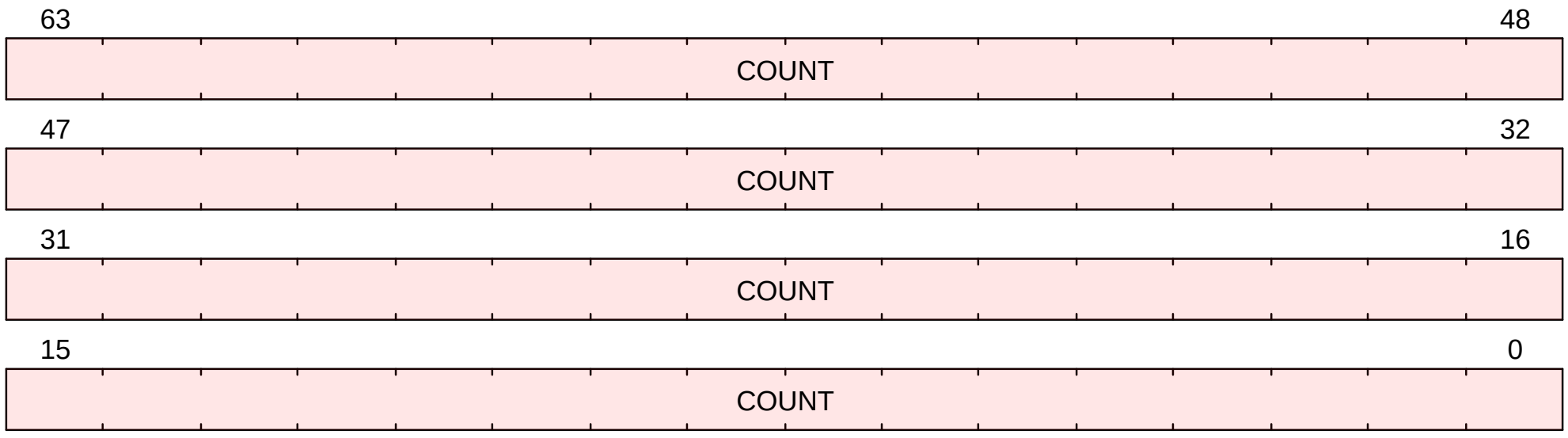


Figure 41. time format

C.41.3. Field Summary

Name	Location	Type	Reset Value
time.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.41.4. Fields

time.COUNT Field

Location:

63:0

Description:

Reports the current wall-clock time from the timer device.

Alias of the `mtime` memory-mapped CSR.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.41.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (!TIME_CSR_IMPLEMENTED) {
  unimplemented_csr($encoding);
}
if (mode() == PrivilegeMode::S) {
  if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.TM & scounteren.TM) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.TM == 1'b0 && mcounteren.TM == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.TM & scounteren.TM) == 1'b0) && (mcounteren.IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mtime();
```

C.42. timeh

High-half timer for RDTIME Instruction

timeh is only defined in RV32.

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR `mtimeh`.

Privilege mode access is controlled with mcounteren.TM, scounteren.TM, and hcounteren.TM as follows:

mcounteren.TM	scounteren.TM	scounteren.TM	time behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	0	0	read-only	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	1	0	read-only	read-only	Illegal Instruction	Illegal Instruction
1	0	1	read-only	Illegal Instruction	read-only	Illegal Instruction
1	1	1	read-only	read-only	read-only	read-only

C.42.1. Attributes

CSR Address	0xc81
Defining extension	Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

C.42.2. Format

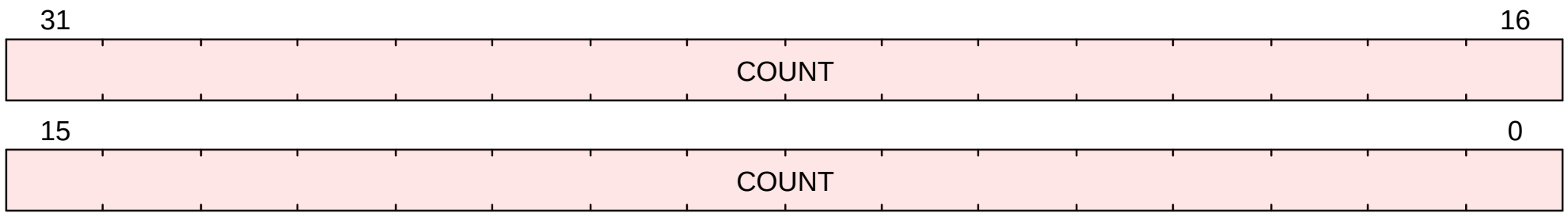


Figure 42. timeh format

C.42.3. Field Summary

Name	Location	Type	Reset Value
time h.COUNT	31:0	RO-H	UNDEFINED_LEGAL

C.42.4. Fields

timeh.COUNT Field

Location:

31:0

Description:

Reports the most significant 32 bits of the current wall-clock time from the timer device.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.42.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (!TIME_CSR_IMPLEMENTED) {
  unimplemented_csr($encoding);
}
if (mode() == PrivilegeMode::S) {
  if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.TM & scounteren.TM) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.TM == 1'b0 && mcounteren.TM == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.TM & scounteren.TM) == 1'b0) && (mcounteren.IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mtime()[63:32];
```

Appendix D: IDL Function Details

D.1. implemented? (generated)

Return true if the implementation supports `extension`.

Return Type	`Boolean`
Arguments	`ExtensionName extension`

D.2. implemented_version? (generated)

Return true if the implementation supports `extension` meeting 'version_requirement'.

Return Type	`Boolean`
Arguments	`ExtensionName extension, String version_requirement`

D.3. implemented_csr? (generated)

Return true if csr_addr is an implemented CSR

Return Type	`Boolean`
Arguments	`Bits<12> csr_addr`

D.4. direct_csr_lookup (generated)

Return CSR info for a CSR with direct address csr_addr.

If no CSR exists, <return_value>.valid == false

Return Type	`Csr`
Arguments	`Bits<12> csr_addr`

D.5. indirect_csr_lookup (generated)

Return CSR info for a CSR with indirect address csr_addr at window slot window_slot.

If no CSR exists, <return_value>.valid == false

Return Type	`Csr`
Arguments	`Bits<MXLEN> csr_addr, Bits<4> window_slot`

D.6. csr_hw_read (generated)

Returns the raw value of csr

Return Type	`Bits`
Arguments	`Csr csr`

D.7. csr_sw_read (generated)

Returns the result of CSR[csr].sw_read(); i.e., the software view of the register

Return Type	`Bits`
Arguments	`Csr csr`

D.8. csr_sw_write (generated)

Writes value to csr, applying an WARL transformations first.

Uses the sw_write(...) functions of CSR field definitions.

Return Type	`void`
Arguments	`Csr csr, Bits<MXLEN> value`

D.9. unpredictable (builtin)

Indicate that the hart has reached a state that is unpredictable because the RISC-V spec allows multiple behaviors. Generally, this will be a fatal condition to any emulation, since it is unclear what to do next.

The single argument *why* is a string describing why the hart entered an unpredictable state.

Return Type	`void`
Arguments	`String why`

D.10. read_hpm_counter (builtin)

Returns the value of hpmcounterN.

N must be between 3..31.

hpmcounterN must be implemented.

Return Type	`Bits`
Arguments	`Bits<5> n`

D.11. hartid (builtin)

Returns the value for mhartid as seen by this hart.

Must obey the rules of the priv spec:

> The mhartid CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

Return Type	`XReg`
Arguments	None

D.12. read_mcycle (builtin)

Return the current value of the cycle counter.

Return Type	`Bits`
Arguments	None

D.13. read_mtime (builtin)

Return the current value of the real time device.

Return Type	`Bits`
Arguments	None

D.14. sw_write_mcycle (builtin)

Given a *value* that software is trying to write into mcycle, perform the write and return the value that will actually be written.

Return Type	`Bits`
Arguments	`Bits<64> value`

D.15. cache_block_zero (builtin)

Zero the cache block at the given physical address.

The cache block may be zeroed using 1 or more writes.

A cache-block-sized region is zeroed regardless of whether or not the memory is in a cacheable PMA region.

Return Type	`void`
Arguments	`XReg cache_block_physical_address`

D.16. eei_ecall_from_m (builtin)

When TRAP_ON_ECALL_FROM_M is false, this function will be called to emulate the EEI handling of ECALL-from-M.

If TRAP_ON_ECALL_FROM_M is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.17. eei_ecall_from_s (builtin)

When TRAP_ON_ECALL_FROM_S is false, this function will be called to emulate the EEI handling of ECALL-from-S.

If TRAP_ON_ECALL_FROM_S is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.18. eei_ecall_from_u (builtin)

When TRAP_ON_ECALL_FROM_U is false, this function will be called to emulate the EEI handling of ECALL-from-U.

If TRAP_ON_ECALL_FROM_U is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.19. eei_ecall_from_vs (builtin)

When TRAP_ON_ECALL_FROM_VS is false, this function will be called to emulate the EEI handling of ECALL-from-VS.

If TRAP_ON_ECALL_FROM_VS is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.20. eei_ebreak (builtin)

When TRAP_ON_EBREAK is false, this function will be called to emulate the EEI handling of EBREAK

If TRAP_ON_EBREAK is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.21. memory_model_acquire (builtin)

Perform an acquire; that is, ensure that no subsequent operation in program order appears to an external observer to occur after the operation calling this function.

Return Type	`void`
Arguments	None

D.22. memory_model_release (builtin)

Perform a release; that is, ensure that no prior store in program order can be observed external to this hart after this function returns.

Return Type	`void`
Arguments	None

D.23. assert (builtin)

Assert that a condition is true. Failure represents an error in the IDL model.

Return Type	`void`
Arguments	`Boolean test, String message`

D.24. notify_mode_change (builtin)

Called whenever the privilege mode changes. Downstream tools can use this to hook events.

Return Type	`void`
Arguments	`PrivilegeMode new_mode, PrivilegeMode old_mode`

D.25. abort_current_instruction (builtin)

Abort the current instruction, and start refetching from $pc.

Return Type	`void`
Arguments	None

D.26. ebreak (builtin)

Raise an `Environment Break` exception, returning control to the debug environment.

Return Type	`void`
Arguments	None

D.27. prefetch_instruction (builtin)

Hint to prefetch a block containing virtual_address for an upcoming fetch.

Return Type	`void`
Arguments	`XReg virtual_address`

D.28. prefetch_read (builtin)

Hint to prefetch a block containing virtual_address for an upcoming load.

Return Type	`void`
Arguments	`XReg virtual_address`

D.29. prefetch_write (builtin)

Hint to prefetch a block containing virtual_address for an upcoming store.

Return Type	`void`
Arguments	`XReg virtual_address`

D.30. fence (builtin)

Execute a memory ordering fence.(according to the FENCE instruction).

Return Type	`void`
Arguments	`Boolean pi, Boolean pr, Boolean po, Boolean pw, Boolean si, Boolean sr, Boolean so, Boolean sw`

D.31. fence_tso (builtin)

Execute a TSO memory ordering fence.(according to the FENCE instruction).

Return Type	`void`
Arguments	None

D.32. ifence (builtin)

Execute a memory ordering instruction fence (according to FENCE.I).

Return Type	`void`
Arguments	None

D.33. pause (builtin)

Pause hart retirement for a implementation-defined period of time, which may be zero.

See Zihintpause for more.

Return Type	`void`
Arguments	None

D.34. pow (generated)

Return `value` to the power `exponent`.

Return Type	`XReg`
Arguments	`XReg value, XReg exponent`

D.35. maybe_cache_translation (generated)

Given a translation result, potentially cache the result for later use. This function models a TLB fill operation. A valid implementation does nothing.

Return Type	`void`
Arguments	`XReg vaddr, MemoryOperation op, TranslationResult result`

D.36. cached_translation (generated)

Possibly returns a cached translation result matching vaddr.

CachedTranslationResult contains a Boolean 'valid' field. If valid, 'result' is a usable translation. Otherwise, the cache lookup failed.

Return Type	`CachedTranslationResult`
Arguments	`XReg vaddr, MemoryOperation op`

D.37. order_pgtbl_writes_before_vmafence (builtin)

Orders all writes prior to this call in global memory order that affect a page table in the set identified by order_type before any subsequent sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.W.INVAL.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType order_type`

D.38. order_pgtbl_reads_after_vmafence (builtin)

Orders all reads after to this call in global memory order to a page table in the set identified by order_type after any prior sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.INVAL.IR.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType order_type`

D.39. invalidate_translations (generated)

Locally invalidate the cached S-mode/VS-mode/G-stage address translations contained in the set identified by inval_type.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType inval_type`

D.40. read_physical_memory

Read from physical memory.

Return Type	`Bits<len>`
Arguments	`XReg paddr`

```
if (len == 8) {
  return read_physical_memory_8(paddr);
} else if (len == 16) {
  return read_physical_memory_16(paddr);
} else if (len == 32) {
  return read_physical_memory_32(paddr);
} else if (len == 64) {
  return read_physical_memory_64(paddr);
} else {
  assert(false, "Invalid len");
}
```

D.41. read_physical_memory_8 (builtin)

Read a byte from physical memory.

Return Type	`Bits⑧`
Arguments	`XReg paddr`

D.42. read_physical_memory_16 (builtin)

Read two bytes from physical memory.

Return Type	`Bits⑯`
Arguments	`XReg paddr`

D.43. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

D.44. read_physical_memory_64 (builtin)

Read eight bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

D.45. write_physical_memory

Write to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<len> value`

```
if (len == 8) {
  write_physical_memory_8(paddr, value);
} else if (len == 16) {
  write_physical_memory_16(paddr, value);
} else if (len == 32) {
  write_physical_memory_32(paddr, value);
} else if (len == 64) {
  write_physical_memory_64(paddr, value);
} else {
  assert(false, "Invalid len");
}
```

D.46. write_physical_memory_8 (builtin)

Write a byte to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<8> value`

D.47. write_physical_memory_16 (builtin)

Write two bytes to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<16> value`

D.48. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<32> value`

D.49. write_physical_memory_64 (builtin)

Write eight bytes to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<64> value`

D.50. wfi (builtin)

Wait-for-interrupt: hint that the processor should enter a low power state until the next interrupt.

A valid implementation is a no-op.

The model will advance the PC; this function does not need to.

Return Type	`void`
Arguments	None

D.51. pma_applies? (builtin)

Checks if *attr* is applied to the entire physical address region between [paddr, paddr + len) based on static PMA attributes.

Return Type	`Boolean`
Arguments	`PmaAttribute attr, Bits<PHYS_ADDR_WIDTH> paddr, U32 len`

D.52. atomic_check_then_write_32 (builtin)

Atomically:

- Reads 32-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 32-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, Bits<32> compare_value, Bits<32> write_value`

D.53. atomic_check_then_write_64 (builtin)

Atomically:

- Reads 64-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, Bits<64> compare_value, Bits<64> write_value`

D.54. atomically_set_pte_a (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' bit and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len`

D.55. atomically_set_pte_a_d (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' and 'D' bits and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len`

D.56. atomic_read_modify_write_32 (builtin)

Atomically read-modify-write 32-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	`Bits`
Arguments	`Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<32> value, AmoOperation op`

D.57. atomic_read_modify_write_64 (builtin)

Atomically read-modify-write 64-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	`Bits`
Arguments	`Bits<PHYS_ADDR_WIDTH>  phys_addr, Bits<64>              value, AmoOperation op`

D.58. set_external_interrupt

Set an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  mip.MEIP = 1'b1;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  pending_smode_external_interrupt = true;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  mip.VSEIP = 1'b1;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.59. clear_external_interrupt

Clear an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  mip.MEIP = 1'b0;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  pending_smode_external_interrupt = false;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  mip.VSEIP = 1'b0;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.60. set_software_interrupt

Set a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  mip.MSIP = 1'b1;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  mip.SSIP = 1'b1;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.61. clear_software_interrupt

Clear a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  mip.MSIP = 1'b0;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  mip.SSIP = 1'b0;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.62. set_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  mip.MTIP = 1'b1;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  mip.STIP = 1'b1;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  pending_vsmode_timer_interrupt = true;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.63. clear_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  mip.MTIP = 1'b0;
} else if ((misa.S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  mip.STIP = 1'b0;
} else if ((misa.H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  pending_vsmode_timer_interrupt = false;
} else {
```

```
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.64. refresh_pending_interrupts

refreshes the calculation of a pending interrupt

needs to be called after any state update that could change a pending interrupt. This includes: - CSR[mip] - CSR[mie] - CSR[mstatus].MIE - CSR[mstatus].SIE - CSR[vsstatus].SIE - CSR[mideleg] - CSR[sideleg] - CSR[hideleg] - CSR[hvip] - CSR[hgeip] - CSR[hgeie] - mode changes

Return Type	`void`
Arguments	None

```
Bits<MXLEN> pending_ints = mip.sw_read() & $bits(mie);
if (pending_ints == 0) {
  pending_and_enabled_interrupts = 0;
  return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") &&
implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_enabled_ints = mode() == PrivilegeMode::M) && (mstatus.MIE == 1'b0 ? 0 : ($bits(mie) & (HAS_MIDELEG ?
~$bits(mideleg) : ~MXLEN'0));
Bits<MXLEN> mmode_pending_and_enabled = pending_ints & mmode_enabled_ints;
if (mmode_pending_and_enabled != 0) {
  pending_and_enabled_interrupts = mmode_pending_and_enabled;
  return ;
}
if (misa.S == 1'b1) {
  Bits<MXLEN> smode_enabled_ints = mode() == PrivilegeMode::M) || (mstatus.SIE == 1'b0 ? 0 : $bits(mie) & ($bits(mideleg));
  Bits<MXLEN> smode_pending_and_enabled = pending_ints & smode_enabled_ints;
  if (smode_pending_and_enabled != 0) {
    pending_and_enabled_interrupts = smode_pending_and_enabled;
    return ;
  }
}
pending_and_enabled_interrupts = 0;
```

D.65. highest_priority_interrupt

Given a bitmask of interrupts in the format of MIE/MIP, return the highest priority interrupt code that is set

Interrupt priority is: MEI, MSI, MTI, SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI

Return Type	`InterruptCode`
Arguments	`Bits<MXLEN> int_mask`

```
if (int_mask[$bits(InterruptCode::MachineExternal)] == 1'b1) {
  return InterruptCode::MachineExternal;
} else if (int_mask[$bits(InterruptCode::MachineSoftware)] == 1'b1) {
  return InterruptCode::MachineSoftware;
} else if (int_mask[$bits(InterruptCode::MachineTimer)] == 1'b1) {
  return InterruptCode::MachineTimer;
}
if (misa.S == 1'b1) {
  if (int_mask[$bits(InterruptCode::SupervisorExternal)] == 1'b1) {
    return InterruptCode::SupervisorExternal;
  } else if (int_mask[$bits(InterruptCode::SupervisorSoftware)] == 1'b1) {
    return InterruptCode::SupervisorSoftware;
  } else if (int_mask[$bits(InterruptCode::SupervisorTimer)] == 1'b1) {
    return InterruptCode::SupervisorTimer;
  }
}
if (implemented?(ExtensionName::Sscofpmf)) {
  if (int_mask[$bits(InterruptCode::LocalCounterOverflow)] == 1'b1) {
```

```
    return InterruptCode::LocalCounterOverflow;
  }
}
assert(false, "There is no valid interrupt");
```

D.66. choose_interrupt

Return the highest priority interrupt that is both pending and enabled and the mode it will be taken in

Return Type	`InterruptCode, PrivilegeMode`
Arguments	None

```
InterruptCode chosen;
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") &&
implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_pending_and_enabled = pending_and_enabled_interrupts & ~(HAS_MIDELEG ? $bits(mideleg) : MXLEN'0);
if (mmode_pending_and_enabled != 0) {
  assert((mode() != PrivilegeMode::M) || (mstatus.MIE == 1'b1), "M-mode interrupts are not enabled");
  chosen = highest_priority_interrupt(mmode_pending_and_enabled);
} else if (misa.S == 1'b1) {
  Bits<MXLEN> smode_pending_and_enabled = (pending_and_enabled_interrupts & $bits(mideleg));
  if (smode_pending_and_enabled != 0) {
    assert((mode() == PrivilegeMode::U) || (mode() == PrivilegeMode::VU) || (mode() == PrivilegeMode::VS) || (mode() ==
PrivilegeMode::S) && (mstatus.SIE == 1'b1), "S-mode interrupt can't be triggered");
    chosen = highest_priority_interrupt(smode_pending_and_enabled);
  }
}
assert($bits(chosen) != 0, "Didn't pick interrupt?");
PrivilegeMode to_mode;
Bits<MXLEN> chosen_mask = (MXLEN'1 << $bits(chosen));
if (((HAS_MIDELEG ? $bits(mideleg) : MXLEN'0) & chosen_mask) == 0) {
  to_mode = PrivilegeMode::M;
} else {
  if (misa.S == 1'b1) {
    to_mode = PrivilegeMode::S;
  } else {
    to_mode = PrivilegeMode::U;
  }
}
return chosen, to_mode;
```

D.67. take_interrupt

Take (adjust CSRs and set PC to handler) the highest priority interrupt that is both pending and enabled

Return Type	`void`
Arguments	None

```
PrivilegeMode to_mode;
InterruptCode code;
(code, to_mode = choose_interrupt());
if (to_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
  mstatus.MPP = $bits(mode())[1:0];
  if (misa.H == 1'b1) {
    if (MXLEN == 64) {
      mstatus.MPV = $bits(mode())[2];
    } else {
      mstatush.MPV = $bits(mode())[2];
    }
    mtval2.VALUE = 0;
    mtinst.VALUE = 0;
  }
  mcause.CODE = $bits(code);
  mcause.INT = 1'b1;
  mtval.VALUE = 0;
```

```
  if (mtvec.MODE == 0) {
    $pc = {mtvec.BASE, 2'b00};
  } else if (mtvec.MODE == 1'b1) {
    $pc = {mtvec.BASE, 2'b00} + ($bits(code) * 4);
  }
} else if ((misa.S == 1'b1) && (to_mode == PrivilegeMode::S)) {
  sepc.PC = $pc;
  mstatus.SPP = $bits(mode())[0];
  if (misa.H == 1'b1) {
    hstatus.SPV = $bits(mode())[2];
  }
  scause.CODE = $bits(code);
  scause.INT = 1'b1;
  stval.VALUE = 0;
  if (stvec.MODE == 0) {
    $pc = {stvec.BASE, 2'b00};
  } else if (stvec.MODE == 1'b1) {
    $pc = {stvec.BASE, 2'b00} + ($bits(code) * 4);
  }
} else if ((misa.H == 1'b1) && (to_mode == PrivilegeMode::VS)) {
  vsepc.PC = $pc;
  vsstatus.SPP = $bits(mode())[0];
  vscause.CODE = $bits(code);
  vscause.INT = 1'b1;
  vstval.VALUE = 0;
  if (vstvec.MODE == 0) {
    $pc = {vstvec.BASE, 2'b00};
  } else if (vstvec.MODE == 1'b1) {
    $pc = {vstvec.BASE, 2'b00} + ($bits(code) * 4);
  }
}
set_mode_no_refresh(to_mode);
```

D.68. fetch_memory_aligned_16

Fetch 16 bits from virtual memory using a known aligned address.

Return Type	`Bits⑯`
Arguments	`XReg virtual_address`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 16, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<16>(result.paddr);
```

D.69. fetch_memory_aligned_32

Fetch 32 bits from virtual memory using a known aligned address.

Return Type	`Bits`
Arguments	`XReg virtual_address`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 32, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
```

```
return read_physical_memory<32>(result.paddr);
```

D.70. power_of_2?

Returns true if value is a power of two, false otherwise

Return Type	Boolean
Arguments	Bits<N> value

```
return (value != 0) && value & (value - 1 == 0);
```

D.71. ary_includes?

Returns true if *value* is an element of ary, and false otherwise

Return Type	Boolean
Arguments	Bits<ELEMENT_SIZE> ary[ARY_SIZE], Bits<ELEMENT_SIZE> value

```
for (U32 i = 0; i < ARY_SIZE; i++) {
  if (ary[i] == value) {
    return true;
  }
}
return false;
```

D.72. has_virt_mem?

Returns true if some virtual memory translation (Sv*) is supported in the config.

Return Type	Boolean
Arguments	None

```
return implemented?(ExtensionName::Sv32) || implemented?(ExtensionName::Sv39) || implemented?(ExtensionName::Sv48) ||
implemented?(ExtensionName::Sv57);
```

D.73. highest_set_bit

Returns the position of the highest (nearest MSB) bit that is '1', or -1 if value is zero.

Return Type	Bits⑧
Arguments	XReg value

```
for (Bits<8> i = xlen() - 1; i >= 0; i--) {
  if (value[i] == 1) {
    return i;
  }
}
return -'sd1;
```

D.74. lowest_set_bit

Returns the position of the lowest (nearest LSB) bit that is '1', or XLEN if value is zero.

Return Type	`Bits⑧`
Arguments	`XReg value`

```
for (Bits<8> i = 0; i < xlen(); i++) {
  if (value[i] == 1) {
    return i;
  }
}
return xlen();
```

D.75. bit_length

Returns the minimum number of bits needed to represent value.

Only works on unsigned values.

The value 0 returns 1.

Return Type	`XReg`
Arguments	`XReg value`

```
for (XReg i = 63; i > 0; i--) {
  if (value[i] == 1) {
    return i;
  }
}
return 1;
```

D.76. count_leading_zeros

Returns the number of leading 0 bits before the most-significant 1 bit of value, or N if value is zero.

Return Type	`Bits<bit_length(N)>`
Arguments	`Bits<N> value`

```
for (U32 i = 0; i < N; i++) {
  if (value[N - 1 - i] == 1) {
    return i;
  }
}
return N;
```

D.77. sext

Sign extend `value` starting at `first_extended_bit`.

Bits [`XLEN-1:`first_extended_bit`] of the return value should get the value of bit (`first_extended bit - 1`).

Return Type	`XReg`
Arguments	`XReg value, XReg first_extended_bit`

```
if (first_extended_bit == MXLEN) {
  return value;
} else {
```

```
  Bits<1> sign = value[first_extended_bit - 1];
  for (U32 i = MXLEN - 1; i >= first_extended_bit; i--) {
    value[i] = sign;
  }
  return value;
}
```

D.78. is_naturally_aligned

Checks if value is naturally aligned to N bits.

Return Type	`Boolean`
Arguments	`XReg value`

```
return true if (N == 8);
XReg Mask = (N / 8) - 1;
return (value & ~Mask) == value;
```

D.79. in_naturally_aligned_region?

Checks if a length-bit access starting at address lies entirely within an N-bit naturally-aligned region.

Return Type	`Boolean`
Arguments	`XReg address, U32  length`

```
XReg Mask = (N / 8) - 1;
return (address & ~Mask) == ((address + length - 1) & ~Mask);
```

D.80. contains?

Given a *region* defined by region_start, region_size, determine if a *target* defined by target_start, target_size is completely contained with the region.

Return Type	`Boolean`
Arguments	`XReg region_start, U32  region_size, XReg target_start, U32  target_size`

```
return target_start >= region_start && (target_start + target_size) <= (region_start + region_size);
```

D.81. set_fp_flag

Add flag to the sticky flags bits in CSR[fcsr]

Return Type	`void`
Arguments	`FpFlag flag`

```
if (flag == FpFlag::NX) {
  fcsr.NX = 1;
} else if (flag == FpFlag::UF) {
  fcsr.UF = 1;
} else if (flag == FpFlag::OF) {
  fcsr.OF = 1;
} else if (flag == FpFlag::DZ) {
  fcsr.DZ = 1;
} else if (flag == FpFlag::NV) {
```

```
  fcsr.NV = 1;
}
```

D.82. rm_to_mode

Convert rm to a RoundingMode.

encoding is the full encoding of the instruction rm comes from.

Will raise an IllegalInstruction exception if rm is a reserved encoding.

Return Type	`RoundingMode`
Arguments	`Bits<3> rm, Bits<32> encoding`

```
if (rm == $bits(RoundingMode::RNE)) {
  return RoundingMode::RNE;
} else if (rm == $bits(RoundingMode::RTZ)) {
  return RoundingMode::RTZ;
} else if (rm == $bits(RoundingMode::RDN)) {
  return RoundingMode::RDN;
} else if (rm == $bits(RoundingMode::RUP)) {
  return RoundingMode::RUP;
} else if (rm == $bits(RoundingMode::RMM)) {
  return RoundingMode::RMM;
} else if (rm == $bits(RoundingMode::DYN)) {
  return $enum(RoundingMode, fcsr.FRM);
} else {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

D.83. mark_f_state_dirty

Potentially updates mstatus.FS to the Dirty (3) state, depending on configuration settings.

Return Type	`void`
Arguments	None

```
if (HW_MSTATUS_FS_DIRTY_UPDATE == "precise") {
  mstatus.FS = 3;
} else if (HW_MSTATUS_FS_DIRTY_UPDATE == "imprecise") {
  unpredictable("The hart may or may not update mstatus.FS now");
}
```

D.84. nan_box

Produces a properly NaN-boxed floating-point value from a floating-point value of smaller size by adding all 1’s to the upper bits.

Return Type	`Bits<TO_SIZE>`
Arguments	`Bits<FROM_SIZE> from_value`

```
assert(FROM_SIZE < TO_SIZE, "Bad template arguments; FROM_SIZE must be less than TO_SIZE");
return {{TO_SIZE - FROM_SIZE{1'b1}}, from_value};
```

D.85. check_f_ok

Checks if instructions from the F extension can be executed, and, if not, raise an exception.

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

```
if (MUTABLE_MISA_F && misa.F == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
if (mstatus.FS == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

D.86. is_sp_neg_inf?

Return true if sp_value is negative infinity.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_NEG_INF;
```

D.87. is_sp_pos_inf?

Return true if sp_value is positive infinity.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_POS_INF;
```

D.88. is_sp_neg_norm?

Returns true if sp_value is a negative normal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 1) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

D.89. is_sp_pos_norm?

Returns true if sp_value is a positive normal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 0) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

D.90. is_sp_neg_subnorm?

Returns true if sp_value is a negative subnormal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 1) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

D.91. is_sp_pos_subnorm?

Returns true if sp_value is a positive subnormal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 0) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

D.92. is_sp_neg_zero?

Returns true if sp_value is negative zero.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_NEG_ZERO;
```

D.93. is_sp_pos_zero?

Returns true if sp_value is positive zero.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_POS_ZERO;
```

D.94. is_sp_nan?

Returns true if sp_value is a NaN (quiet or signaling)

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22:0] != 0);
```

D.95. is_sp_signaling_nan?

Returns true if sp_value is a signaling NaN

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 0) && (sp_value[21:0] != 0);
```

D.96. is_sp_quiet_nan?

Returns true if sp_value is a quiet NaN

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 1);
```

D.97. softfloat_shiftRightJam32

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned. The value of dist can be arbitrarily large. In particular, if dist is greater than 32, the result will be either 0 or 1, depending on whether a is zero or nonzero.

Return Type	`Bits`
Arguments	`Bits<32> a, Bits<32> dist`

```
return (dist < 31) ? a >> dist | (a << (-dist & 31 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

D.98. softfloat_shiftRightJam64

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned.
The value of 'dist' can be arbitrarily large. In particular, if dist is greater than 64, the result will be either 0 or 1, depending on whether a is zero or nonzero.

Return Type	`Bits`
Arguments	`Bits<64> a, Bits<32> dist`

```
return (dist < 63) ? a >> dist | (a << (-dist & 63 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

D.99. softfloat_roundToI32

Round to signed 32-bit integer, using rounding_mode

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
    roundIncrement = 0xFFF;
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 's0) != (sign == 1) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

D.100. softfloat_roundToUI32

Round to unsigned 32-bit integer, using rounding_mode

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1) {
    if (sig == 0) {
      return 0;
    }
    if (roundingMode == RoundingMode::RDN) {
      set_fp_flag(FpFlag::NV);
    }
  } else {
    if (roundingMode == RoundingMode::RUP) {
      roundIncrement = 0xFFF;
    }
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? UI32_NEG_OVERFLOW : UI32_POS_OVERFLOW;
}
Bits<32> z = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  z = z & ~32'b1;
}
if ((z != 0) && (sign == 1)) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? UI32_NEG_OVERFLOW : UI32_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

D.101. packToF32UI

Pack components into a 32-bit value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig`

```
return {sign, exp, sig};
```

D.102. packToF16UI

Pack components into a 16-bit value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<5> exp, Bits<10> sig`

```
return {sign, exp, sig};
```

D.103. softfloat_normSubnormalF16Sig

normalize subnormal half-precision value

Return Type	`Bits<5>, Bits⑩`
Arguments	`Bits<16> hp_value`

```
Bits<8> shift_dist = count_leading_zeros<16>(hp_value);
return 1 - shift_dist, hp_value << shift_dist;
```

D.104. softfloat_roundPackToF32

Round FP value according to mdode and then pack it in IEEE format.

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
  roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
  if ($signed(exp) < 's0) {
    Boolean isTiny = ($signed(exp) < -8's1) || (sig + roundIncrement < 0x80000000);
    sig = softfloat_shiftRightJam32(sig, -exp);
    exp = 0;
    roundBits = sig & 0x7F;
    if (isTiny && (roundBits != 0)) {
      set_fp_flag(FpFlag::UF);
    }
  } else if ('shFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
    set_fp_flag(FpFlag::OF);
    set_fp_flag(FpFlag::NX);
    return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0) ? 1 : 0);   } } sig = (sig + roundIncrement);
```

```
if (sig == 0) {
  exp = 0;
}
return packToF32UI(sign, exp, sig);
```

D.105. softfloat_normRoundPackToF32

Normalize, round, and pack into a 32-bit floating point value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
  return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
  return softfloat_roundPackToF32(sign, exp, sig << shiftDist, mode);
}
```

D.106. signF32UI

Extract sign-bit of a 32-bit floating point number

Return Type	`Bits①`
Arguments	`Bits<32> a`

```
return a[31];
```

D.107. expF32UI

Extract exponent of a 32-bit floating point number

Return Type	`Bits⑧`
Arguments	`Bits<32> a`

```
return a[30:23];
```

D.108. fracF32UI

Extract significand of a 32-bit floating point number

Return Type	`Bits`
Arguments	`Bits<32> a`

```
return a[22:0];
```

D.109. returnNonSignalingNaN

Returns a non-signalling NaN version of the floating-point number Does not modify the input

Return Type	`U32`
Arguments	`U32 a`

```
U32 a_copy = a;
a_copy[22] = 1'b1;
return a_copy;
```

D.110. returnMag

Returns magnitude of the given number Does not modify the input

Return Type	`U32`
Arguments	`U32 a`

```
U32 a_copy = a;
a_copy[31] = 1'b0;
return a_copy;
```

D.111. returnLargerMag

Returns the larger number between a and b by magnitude If either number is signaling NaN then that is made quiet

Return Type	`U32`
Arguments	`U32 a, U32 b`

```
U32 mag_a = returnMag(a);
U32 mag_b = returnMag(b);
U32 nonsig_a = returnNonSignalingNaN(a);
U32 nonsig_b = returnNonSignalingNaN(b);
if (mag_a < mag_b) {
  return nonsig_b;
}
if (mag_b < mag_a) {
  return nonsig_a;
}
return (nonsig_a < nonsig_b) ? nonsig_a : nonsig_b;
```

D.112. softfloat_propagateNaNF32UI

Interpreting 'a' and 'b' as the bit patterns of two 32-bit floating- | point values, at least one of which is a NaN, returns the bit pattern of | the combined NaN result. If either 'a' or 'b' has the pattern of a | signaling NaN, the invalid exception is raised.

Return Type	`U32`
Arguments	`U32 a, U32 b`

```
Boolean isSigNaN_a = is_sp_signaling_nan?(a);
Boolean isSigNaN_b = is_sp_signaling_nan?(b);
if (isSigNaN_a || isSigNaN_b) {
  set_fp_flag(FpFlag::NV);
}
return SP_CANONICAL_NAN;
```

D.113. softfloat_addMagsF32

Returns sum of the magnitudes of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'd0) {
    z = a + b;
    return z;
  }
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  signZ = signF32UI(a);
  expZ = expA;
  sigZ = 32'h01000000 + sigA + sigB;
  if (sigZ & 0x1) == 0) && (expZ < 8'hFE {
    sigZ = sigZ >> 1;
    return (32'h0 + (signZ << 31) + (expZ << 23) + sigZ);
  }
  sigZ = sigZ << 6;
} else {
  signZ = signF32UI(a);
  U32 sigA_32 = 32'h0 + (sigA << 6);
  U32 sigB_32 = 32'h0 + (sigA << 6);
  if (expDiff < 0) {
    if (expB == 8'hFF) {
      if (sigB != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, 8'hFF, 23'h0);
    }
    expZ = expB;
    sigA_32 = (expA == 0) ? 2 * sigA_32 : (sigA_32 + 0x20000000);
    sigA_32 = softfloat_shiftRightJam32(sigA_32, (32'h0 - expDiff));
  } else {
    if (expA == 8'hFF) {
      if (sigA != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
    }
    expZ = expA;
    sigB_32 = (expB == 0) ? 2 * sigB_32 : (sigB_32 + 0x20000000);
    sigB_32 = softfloat_shiftRightJam32(sigB_32, (32'h0 + expDiff));
  }
  U32 sigZ = 0x20000000 + sigA + sigB;
  if (sigZ < 0x40000000) {
    expZ = expZ - 1;
    sigZ = sigZ << 1;
  }
}
return softfloat_roundPackToF32(signZ, expZ, sigZ[22:0], mode);
```

D.114. softfloat_subMagsF32

Returns difference of the magnitudes of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
U32 sigDiff;
U32 sigX;
U32 sigY;
U32 sigA_32;
U32 sigB_32;
Bits<8> shiftDist;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  sigDiff = sigA - sigB;
  if (sigDiff == 0) {
    return packToF32UI(((mode == RoundingMode::RDN) ? 1 : 0), 0, 0);
  }
  if (expA != 0) {
    expA = expA - 1;
  }
  signZ = signF32UI(a);
  if (sigDiff < 0) {
    signZ = ~signZ;
    sigDiff = -32'sh1 * sigDiff;
  }
  shiftDist = count_leading_zeros<32>(sigDiff) - 8;
  expZ = expA - shiftDist;
  if (expZ < 0) {
    shiftDist = expA;
    expZ = 0;
  }
  return packToF32UI(signZ, expZ, sigDiff << shiftDist);
} else {
  signZ = signF32UI(a);
  sigA_32 = 32'h0 + (sigA << 7);
  sigB_32 = 32'h0 + (sigB << 7);
  if (expDiff < 0) {
    signZ = ~signZ;
    if (expB == 0xFF) {
      if (sigB_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, expB, 0);
    }
    expZ = expB - 1;
    sigX = sigB_32 | 0x40000000;
    sigY = sigA_32 + ((expA != 0) ? 0x40000000 : sigA_32);
    expDiff = -expDiff;
  } else {
    if (expA == 0xFF) {
      if (sigA_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
```

```
    }
    expZ = expA - 1;
    sigX = sigA_32 | 0x40000000;
    sigY = sigB_32 + ((expB != 0) ? 0x40000000 : sigB_32);
  }
  return softfloat_normRoundPackToF32(signZ, expZ, sigX - softfloat_shiftRightJam32(sigY, expDiff), mode);
}
```

D.115. f32_add

Returns sum of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_subMagsF32(a, b, mode);
} else {
  return softfloat_addMagsF32(a, b, mode);
}
```

D.116. f32_sub

Returns difference of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_addMagsF32(a, b, mode);
} else {
  return softfloat_subMagsF32(a, b, mode);
}
```

D.117. i32_to_f32

Converts 32-bit signed integer to 32-bit floating point number

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFFF) == 0) {
  return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32(sign, 0x9C, magnitude_of_A, mode);
```

D.118. ui32_to_f32

Converts 32-bit unsigned integer to 32-bit floating point number

Return Type	`U32`

Arguments	`U32 a, RoundingMode mode`

```
if (a == 0) {
  return a;
}
if (a[31] == 1) {
  return softfloat_roundPackToF32(0, 0x9D, a >> 1 | (a & 1), mode);
} else {
  return softfloat_normRoundPackToF32(0, 0x9C, a, mode);
}
```

D.119. f32_to_i32

Converts 32-bit floating point number to a signed 32-bit integer

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  return I32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToI32(sign, sig64, mode);
```

D.120. f32_to_ui32

Converts 32-bit floating point number to an unsigned 32-bit integer

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  return UI32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
```

```
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToUI32(sign, sig64, mode);
```

D.121. softfloat_roundPackToF32_no_flag

Round FP value according to mdode and then pack it in IEEE format. No flags to be set

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
  roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
  if ($signed(exp) < 's0) {
    Boolean isTiny = ($signed(exp) < -8's1) || (sig + roundIncrement < 0x80000000);
    sig = softfloat_shiftRightJam32(sig, -exp);
    exp = 0;
    roundBits = sig & 0x7F;
  } else if ('shFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
    return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0) ? 1 : 0);   } } sig = (sig + roundIncrement);
if (sig == 0) {
  exp = 0;
}
return packToF32UI(sign, exp, sig);
```

D.122. softfloat_normRoundPackToF32_no_flag

Normalize, round, and pack into a 32-bit floating point value No flags to be set

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
  return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
  return softfloat_roundPackToF32_no_flag(sign, exp, sig << shiftDist, mode);
}
```

D.123. i32_to_f32_no_flag

Converts 32-bit signed integer to 32-bit floating point number No flags to be set

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFFF) == 0) {
  return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32_no_flag(sign, 0x9C, magnitude_of_A, mode);
```

D.124. softfloat_roundToI32_no_flag

Round to signed 32-bit integer, using rounding_mode No flag to be set

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
    roundIncrement = 0xFFF;
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 's0) != (sign == 1) {
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
return z;
```

D.125. f32_to_i32_no_flag

Converts 32-bit floating point number to a signed 32-bit integer No flags to be set

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  return I32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToI32_no_flag(sign, sig64, mode);
```

D.126. round_f32_to_integral

Rounds 32-bit floating point number to a signed 32-bit integer. This 32-bit integer is represented as a floating point number and returned.

Return Type	`U32`

Arguments	`U32 a, RoundingMode mode`

```
if ((is_sp_neg_inf?(a)) || (is_sp_pos_inf?(a)) || (is_sp_pos_zero?(a)) || (is_sp_neg_zero?(a))) {
  return a;
} else if (is_sp_signaling_nan?(a)) {
  set_fp_flag(FpFlag::NV);
  return a;
}
U32 intermediate;
intermediate = f32_to_i32_no_flag(a, mode);
return i32_to_f32_no_flag(intermediate, mode);
```

D.127. mode

Returns the current active privilege mode.

Return Type	`PrivilegeMode`
Arguments	None

```
if ((!implemented?(ExtensionName::S)) && (!implemented?(ExtensionName::U)) && (!implemented?(ExtensionName::H))) {
  return PrivilegeMode::M;
} else {
  return current_mode;
}
```

D.128. set_mode_no_refresh

Set the current privilege mode to `new_mode`, but don't refresh interrupts

Return Type	`void`
Arguments	`PrivilegeMode new_mode`

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
}
```

D.129. set_mode

Set the current privilege mode to `new_mode`

Return Type	`void`
Arguments	`PrivilegeMode new_mode`

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
  refresh_pending_interrupts();
}
```

D.130. compatible_mode?

Returns true if target_mode is more privileged than actual_mode.

Return Type	Boolean
Arguments	PrivilegeMode target_mode, PrivilegeMode actual_mode

```
if (target_mode == PrivilegeMode::M) {
  return actual_mode == PrivilegeMode::M;
} else if (target_mode == PrivilegeMode::S) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S);
} else if (target_mode == PrivilegeMode::U) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::U);
} else if (target_mode == PrivilegeMode::VS) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS);
} else if (target_mode == PrivilegeMode::VU) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS) ||
(actual_mode == PrivilegeMode::VU);
}
```

D.131. exception_handling_mode

Returns the target privilege mode that will handle synchronous exception `exception_code`

Return Type	PrivilegeMode
Arguments	ExceptionCode exception_code

```
if (mode() == PrivilegeMode::M) {
  return PrivilegeMode::M;
} else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::HS) || (mode() == PrivilegeMode::U) {
  if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
    return PrivilegeMode::HS;
  } else {
    return PrivilegeMode::M;
  }
} else {
  assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU, "Unexpected mode");
  if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
    if (($bits(hedeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
      return PrivilegeMode::VS;
    } else {
      return PrivilegeMode::HS;
    }
  } else {
    return PrivilegeMode::M;
  }
}
```

D.132. creg2reg

Maps a C register index (*e.g.*, `rs1'` in the specification) to an X register index. From the specification:

Table 16. Registers specified by the three-bit rs1', rs2', and rd' fields of the CIW, CL, CS, CA, and CB formats.

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

Return Type	Bits⑤
Arguments	Bits<3> creg_idx

```
return {2'b01, creg_idx};
```

D.133. unimplemented_csr

Either raises an IllegalInstruction exception or enters unpredictable state, depending on the setting of the TRAP_ON_UNIMPLEMENTED_CSR parameter.

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

```
if (TRAP_ON_UNIMPLEMENTED_CSR) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else {
  unpredictable("Accessing an unimplmented CSR");
}
```

D.134. mtval_readonly?

Returns whether or not CSR[mtval] is read-only based on implementation options

Return Type	`Boolean`
Arguments	None

```
return !(REPORT_VA_IN_MTVAL_ON_BREAKPOINT || REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED || REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED || REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT || REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT
|| REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_MTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_MTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
```

D.135. stval_readonly?

Returns whether or not CSR[stval] is read-only based on implementation options

Return Type	`Boolean`
Arguments	None

```
if (implemented?(ExtensionName::S)) {
  return !(REPORT_VA_IN_STVAL_ON_BREAKPOINT || REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED
|| REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ||
REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ||
REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT
|| REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_STVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_STVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
  return true;
}
```

D.136. vstval_readonly?

Returns whether or not CSR[vstval] is read-only based on implementation options

Return Type	`Boolean`
Arguments	None

```
if (implemented?(ExtensionName::H)) {
```

```
  return !(REPORT_VA_IN_VSTVAL_ON_BREAKPOINT || REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_VSTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_VSTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
  return true;
}
```

D.137. mtval_for

Given an exception code and a **legal** non-zero value for mtval, returns the value to be written in mtval considering implementation options

Return Type	`XReg`
Arguments	`ExceptionCode exception_code, XReg tval`

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_MTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

D.138. stval_for

Given an exception code and a **legal** non-zero value for stval, returns the value to be written in stval considering implementation options

Return Type	`XReg`
Arguments	`ExceptionCode exception_code, XReg tval`

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_STVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
```

```
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

D.139. vstval_for

Given an exception code and a **legal** non-zero value for vstval, returns the value to be written in vstval considering implementation options

Return Type	`XReg`
Arguments	`ExceptionCode exception_code, XReg tval`

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_VSTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

D.140. raise_guest_page_fault

Raise a guest page fault exception.

Return Type	`void`
Arguments	`MemoryOperation op, XReg gpa, XReg gva, XReg tinst_value, PrivilegeMode from_mode`

```
ExceptionCode code;
Boolean write_gpa_in_tval;
if (op == MemoryOperation::Read) {
  code = ExceptionCode::LoadGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_LOAD_GUEST_PAGE_FAULT;
```

```
} else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
  code = ExceptionCode::StoreAmoGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_STORE_AMO_GUEST_PAGE_FAULT;
} else {
  assert(op == MemoryOperation::Fetch, "unexpected memory operation");
  code = ExceptionCode::InstructionGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_INSTRUCTION_GUEST_PAGE_FAULT;
}
PrivilegeMode handling_mode = exception_handling_mode(code);
if (handling_mode == PrivilegeMode::S) {
  htval.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  htinst.VALUE = tinst_value;
  sepc.PC = $pc;
  if (!stval_readonly?()) {
    stval.VALUE = stval_for(code, gva);
  }
  $pc = {stvec.BASE, 2'b00};
  scause.INT = 1'b0;
  scause.CODE = $bits(code);
  hstatus.GVA = 1;
  hstatus.SPV = 1;
  hstatus.SPVP = $bits(from_mode)[0];
  mstatus.SPP = $bits(from_mode)[0];
} else {
  assert(handling_mode == PrivilegeMode::M, "unexpected privilege mode");
  mtval2.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  mtinst.VALUE = tinst_value;
  mstatus.MPP = $bits(from_mode)[1:0];
  if (MXLEN == 64) {
    mstatus.MPV = 1;
  } else {
    mstatush.MPV = 1;
  }
}
set_mode(handling_mode);
abort_current_instruction();
```

D.141. raise

Raise synchronous exception number `exception_code`.

The exception may be imprecise, and will cause execution to enter an unpredictable state, if PRECISE_SYNCHRONOUS_EXCEPTIONS is false.

Otherwise, the exception will be precise.

Return Type	`void`
Arguments	`ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval`

```
if (!PRECISE_SYNCHRONOUS_EXCEPTIONS) {
  unpredictable("Imprecise synchronous exception");
} else {
  raise_precise(exception_code, from_mode, tval);
}
```

D.142. raise_precise

Raise synchronous exception number `exception_code`.

Return Type	`void`
Arguments	`ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval`

```
PrivilegeMode handling_mode = exception_handling_mode(exception_code);
if (handling_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
```

```
if (!mtval_readonly?()) {
  mtval.VALUE = mtval_for(exception_code, tval);
}
$pc = {mtvec.BASE, 2'b00};
mcause.INT = 1'b0;
mcause.CODE = $bits(exception_code);
if (misa.H == 1) {
  mtval2.VALUE = 0;
  mtinst.VALUE = 0;
  if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
    if (MXLEN == 32) {
      mstatush.MPV = 1;
    } else {
      mstatus.MPV = 1;
    }
  } else {
    if (MXLEN == 32) {
      mstatush.MPV = 0;
    } else {
      mstatus.MPV = 0;
    }
  }
}
mstatus.MPP = $bits(from_mode);
} else if (misa.S == 1 && (handling_mode == PrivilegeMode::S)) {
  sepc.PC = $pc;
  if (!stval_readonly?()) {
    stval.VALUE = stval_for(exception_code, tval);
  }
  $pc = {stvec.BASE, 2'b00};
  scause.INT = 1'b0;
  scause.CODE = $bits(exception_code);
  mstatus.SPP = $bits(from_mode)[0];
  if (misa.H == 1) {
    htval.VALUE = 0;
    htinst.VALUE = 0;
    hstatus.SPV = $bits(from_mode)[2];
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      hstatus.SPV = 1;
      if (exception_code == ExceptionCode::Breakpoint) && (REPORT_VA_IN_STVAL_ON_BREAKPOINT || exception_code ==
ExceptionCode::LoadAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || exception_code ==
ExceptionCode::StoreAmoAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || exception_code ==
ExceptionCode::InstructionAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || exception_code ==
ExceptionCode::LoadAccessFault) && (REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || exception_code ==
ExceptionCode::StoreAmoAccessFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || exception_code ==
ExceptionCode::InstructionAccessFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || exception_code ==
ExceptionCode::LoadPageFault) && (REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || exception_code == ExceptionCode::StoreAmoPageFault) &&
(REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || exception_code == ExceptionCode::InstructionPageFault) &&
(REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT) {
        hstatus.GVA = 1;
      } else {
        hstatus.GVA = 0;
      }
      hstatus.SPVP = $bits(from_mode)[0];
    } else {
      hstatus.SPV = 0;
      hstatus.GVA = 0;
    }
  }
} else if (misa.H == 1 && (handling_mode == PrivilegeMode::VS)) {
  vsepc.PC = $pc;
  if (!vstval_readonly?()) {
    vstval.VALUE = vstval_for(exception_code, tval);
  }
  $pc = {vstvec.BASE, 2'b00};
  vscause.INT = 1'b0;
  vscause.CODE = $bits(exception_code);
  vsstatus.SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();
```

D.143. ialign

Returns IALIGN, the smallest instruction encoding size, in bits.

Return Type	`Bits⑥`
Arguments	None

```
if (implemented?(ExtensionName::C) && (misa.C == 0x1)) {
  return 16;
} else {
  return 32;
}
```

D.144. jump

Jump to virtual address `target_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

Return Type	`void`
Arguments	`XReg target_addr`

```
if ((ialign() == 16) && target_addr & 0x1) != 0 {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
} else if ((ialign() == 32) && (target_addr & 0x3) != 0) {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
}
$pc = target_addr;
```

D.145. jump_halfword

Jump to virtual halfword address `target_hw_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

Return Type	`void`
Arguments	`XReg target_hw_addr`

```
assert((target_hw_addr & 0x1) == 0x0, "Expected halfword-aligned address in jump_halfword");
if (ialign() != 16) {
  if ((target_hw_addr & 0x3) != 0) {
    raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_hw_addr);
  }
}
$pc = target_hw_addr;
```

D.146. valid_interrupt_code?

Returns true if *code* is a legal interrupt number.

Return Type	`Boolean`
Arguments	`XReg code`

```
if (code > 1 `<< $enum_element_size(InterruptCode - 1)) {
  return false;
```

```
}
if (ary_includes?<$enum_size(InterruptCode), $enum_element_size(InterruptCode)>($enum_to_a(InterruptCode), code)) {
  return true;
} else {
  return false;
}
```

D.147. valid_exception_code?

Returns true if *code* is a legal exception number.

Return Type	Boolean
Arguments	XReg code

```
if (code > 1 `<< $enum_element_size(ExceptionCode - 1)) {
  return false;
}
if (ary_includes?<$enum_size(InterruptCode), $enum_element_size(InterruptCode)>($enum_to_a(InterruptCode), code)) {
  return true;
} else {
  return false;
}
```

D.148. xlen

Returns the effective XLEN for the current privilege mode.

Return Type	Bits⑧
Arguments	None

```
if (MXLEN == 32) {
  return 32;
} else {
  if (mode() == PrivilegeMode::M) {
    if (misa.MXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (misa.MXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    if (mstatus.SXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (mstatus.SXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    if (mstatus.UXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (mstatus.UXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    if (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (hstatus.VSXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU) {
    if (vsstatus.UXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (vsstatus.UXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    }
  }
```

```
}
```

D.149. virtual_mode?

Returns True if the current mode is virtual (VS or VU).

Return Type	`Boolean`
Arguments	None

```
return (mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU);
```

D.150. mask_eaddr

Mask upper N bits of an effective address if pointer masking is enabled

Return Type	`XReg`
Arguments	`XReg eaddr`

```
return eaddr;
```

D.151. pmp_match_64

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 8) * 2;
  Bits<6> shamt = (i % 8) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<64> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_hi = 0;
  Bits<PHYS_ADDR_WIDTH> range_lo = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_lo = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_lo = (pmpaddr_value & ~mask) << 2;
    Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;
    range_hi = pmpaddr_value & ~mask) + len) << 2;   } else if (cfg.A == $bits(PmpCfg_A::NA4 {
    range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
    range_hi = range_lo + 4;
  }
  if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
    return PmpMatchResult::FullMatch, cfg;
```

```
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

D.152. pmp_match_32

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 4);
  Bits<6> shamt = (i % 4) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<32> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_base = 0;
  Bits<PHYS_ADDR_WIDTH> range_limit = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_base = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_base = csr_sw_read(tor_pmpaddr_csr)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_limit = (pmpaddr_csr_value)[PHYS_ADDR_WIDTH - 1:0] - 1;
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 1> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 1> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_base = pmpaddr_value & ~mask;
    range_limit = range_base + mask;
  } else if (cfg.A == $bits(PmpCfg_A::NA4)) {
    range_base = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 1:0];
    range_limit = range_base + 3;
  }
  if (paddr {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

D.153. pmp_match

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
if (MXLEN == 64) {
  return pmp_match_64(paddr, access_size);
} else {
```

```
  return pmp_match_32(paddr, access_size);
}
```

D.154. mpv

Returns the current value of CSR[mstatus].MPV (when MXLEN == 64) of CSR[mstatush].MPV (when MXLEN == 32)

Return Type	`Bits①`
Arguments	None

```
if (implemented?(ExtensionName::H)) {
  return (MXLEN == 32) ? mstatush.MPV : mstatus.MPV;
} else {
  assert(false, "TODO");
}
```

D.155. effective_ldst_mode

Returns the effective privilege mode for normal explicit loads and stores, taking into account the current actual privilege mode and modifications from mstatus.MPRV.

Return Type	`PrivilegeMode`
Arguments	None

```
if (mode() == PrivilegeMode::M) {
  if (misa.U == 1 && mstatus.MPRV == 1) {
    if (mstatus.MPP == 0b00) {
      if (misa.H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VU;
      } else {
        return PrivilegeMode::U;
      }
    } else if (misa.S == 1 && mstatus.MPP == 0b01) {
      if (misa.H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VS;
      } else {
        return PrivilegeMode::S;
      }
    }
  }
}
return mode();
```

D.156. pmp_check

Given a physical address and operation type, return whether or not the access is allowed by PMP.

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, MemoryOperation type`

```
PrivilegeMode mode = effective_ldst_mode();
PmpMatchResult match_result;
PmpCfg cfg;
(match_result, cfg = pmp_match(paddr, access_size));
if (match_result == PmpMatchResult::FullMatch) {
  if (mode == PrivilegeMode::M && (cfg.L == 0)) {
    return true;
  }
  if (type == MemoryOperation::Write && (cfg.W == 0)) {
    return false;
  } else if (type == MemoryOperation::Read && (cfg.R == 0)) {
```

```
    return false;
  } else if (type == MemoryOperation::Fetch && (cfg.X == 0)) {
    return false;
  }
} else if (match_result == PmpMatchResult::NoMatch) {
  if (mode == PrivilegeMode::M) {
    return true;
  } else {
    return false;
  }
} else {
  assert(match_result == PmpMatchResult::PartialMatch, "PMP matching logic error");
  return false;
}
return true;
```

D.157. access_check

Checks if the physical address paddr is able to access memory, and raises the appropriate exception if not.

Return Type	`void`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, XReg vaddr, MemoryOperation type, ExceptionCode fault_type, PrivilegeMode from_mode`

```
if (paddr > 1 `<< PHYS_ADDR_WIDTH) - access_size {
  raise(fault_type, from_mode, vaddr);
}
if (implemented?(ExtensionName::Smpmp)) {
  if (!pmp_check(paddr[PHYS_ADDR_WIDTH - 1:0], access_size, type)) {
    raise(fault_type, from_mode, vaddr);
  }
}
```

D.158. base32?

return True iff current effective XLEN == 32

Return Type	`Boolean`
Arguments	None

```
if (MXLEN == 32) {
  return true;
} else {
  XRegWidth xlen32 = XRegWidth::XLEN32;
  if (mode() == PrivilegeMode::M) {
    return misa.MXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    return mstatus.SXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    return mstatus.UXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    return hstatus.VSXL == $bits(xlen32);
  } else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU, "Unexpected mode");
    return vsstatus.UXL == $bits(xlen32);
  }
}
```

D.159. base64?

return True iff current effective XLEN == 64

Return Type	`Boolean`
Arguments	None

```
return xlen() == 64;
```

D.160. current_translation_mode

Returns the current first-stage translation mode for an explicit load or store from mode given the machine state (e.g., value of satp or vsatp csr).

Returns SatpMode::Reserved if the setting found in satp or vsatp is invalid.

Return Type	`SatpMode`
Arguments	`PrivilegeMode mode`

```
PrivilegeMode effective_mode = effective_ldst_mode();
if (effective_mode == PrivilegeMode::M) {
  return SatpMode::Bare;
}
if (misa.H == 1'b1) {
  if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU) {
    Bits<4> mode_val = vsatp.MODE;
    if (mode_val == $bits(SatpMode::Bare)) {
      return SatpMode::Bare;
    } else if (mode_val == $bits(SatpMode::Sv32)) {
      if (MXLEN == 64) {
        if ((effective_mode == PrivilegeMode::VS) && (hstatus.VSXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
        if ((effective_mode == PrivilegeMode::VU) && (vsstatus.UXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
      }
      if (!SV32_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv32;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV39_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv39;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV48_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv48;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV57_VSMODE_TRANSLATION) {
```

```
        return SatpMode::Reserved;
      }
      return SatpMode::Sv57;
    } else {
      return SatpMode::Reserved;
    }
  } else {
    return SatpMode::Reserved;
  }
} else if (misa.S == 1'b1) {
  assert(effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U, "unexpected priv mode");
  Bits<4> mode_val = satp.MODE;
  if (mode_val == $bits(SatpMode::Bare)) {
    return SatpMode::Bare;
  } else if (mode_val == $bits(SatpMode::Sv32)) {
    if (MXLEN == 64) {
      if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
    }
    if (!implemented?(ExtensionName::Sv32)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv32;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv39)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv39;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv48)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv48;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv57)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv57;
  } else {
    return SatpMode::Reserved;
  }
} else {
  return SatpMode::Reserved;
}
```

D.161. current_gstage_translation_mode

Returns the current second-stage translation mode for a load or store from VS-mode or VU-mode.

Return Type	`HgatpMode`
Arguments	None

```
return $enum(HgatpMode, hgatp.MODE);
```

D.162. translate_gstage

Translates a guest physical address to a physical address.

Return Type	`TranslationResult`
Arguments	`XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U) {
  result.paddr = gpaddr;
  return result;
}
Boolean mxr = mstatus.MXR == 1;
if (GSTAGE_MODE_BARE && hgatp.MODE == $bits(HgatpMode::Bare)) {
  result.paddr = gpaddr;
  return result;
} else if (SV32X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv32x4)) {
  return gstage_page_walk<32, 34, 32, 2>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV39X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv39x4)) {
  return gstage_page_walk<39, 56, 64, 3>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV48X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv48x4)) {
  return gstage_page_walk<48, 56, 64, 4>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV57X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv57x4)) {
  return gstage_page_walk<57, 56, 64, 5>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else {
  if (op == MemoryOperation::Read) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else {
    assert(op == MemoryOperation::Fetch, "unexpected memory op");
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  }
}
```

D.163. tinst_value_for_guest_page_fault

Returns the value of htinst/mtinst for a Guest Page Fault

Return Type	`XReg`
Arguments	`MemoryOperation op, Bits<INSTR_ENC_SIZE> encoding, Boolean for_final_vs_pte`

```
if (for_final_vs_pte) {
  if (op == MemoryOperation::Fetch) {
    if (TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else {
      assert(TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always pseudoinstruction", "Instruction guest page faults can
only report zero/pseudo instruction in tval");
      return 0x00002000;
    }
  } else if (op == MemoryOperation::Read) {
    if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always pseudoinstruction") {
```

```
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002000;
      } else {
        return 0x00003000;
      }
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002020;
      } else {
        return 0x00003020;
      }
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  } else {
    if (REPORT_GPA_IN_TVAL_ON_INTERMEDIATE_GUEST_PAGE_FAULT) {
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002000;
      } else if ((VSXLEN == 64) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN64)) {
        return 0x00003000;
      }
    }
  }
```

D.164. tinst_transform

Returns the standard transformation of an encoding for htinst/mtinst

Return Type	`Bits<INSTR_ENC_SIZE>`
Arguments	`Bits<INSTR_ENC_SIZE> encoding, Bits<5> addr_offset`

```
if (encoding[1:0] == 0b11) {
  if (encoding[6:2] == 5'b00001) {
    return {{12{1'b0}}, addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b01000) {
    return {{7{1'b0}}, encoding[24:20], addr_offset, encoding[14:12], {5{1'b0}}, encoding[6:0]};
  } else if (encoding[6:2] == 5'b01011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b00011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else {
    assert(false, "Bad transform");
  }
} else {
  assert(false, "TODO: compressed instruction");
}
```

D.165. transformed_standard_instruction_for_tinst

Transforms an instruction encoding for htinst.

Return Type	`Bits<INSTR_ENC_SIZE>`
Arguments	`Bits<INSTR_ENC_SIZE> original`

```
assert(false, "TODO");
return 0;
```

D.166. tinst_value

Returns the value of htinst/mtinst for the given exception code.

Return Type	XReg
Arguments	ExceptionCode code, Bits<INSTR_ENC_SIZE> encoding

```
if (code == ExceptionCode::InstructionAddressMisaligned) {
  if (TINST_VALUE_ON_INSTRUCTION_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to an InstructionAddressMisaligned exception");
  }
} else if (code == ExceptionCode::InstructionAccessFault) {
  return 0;
} else if (code == ExceptionCode::IllegalInstruction) {
  return 0;
} else if (code == ExceptionCode::Breakpoint) {
  if (TINST_VALUE_ON_BREAKPOINT == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a Breakpoint exception");
  }
} else if (code == ExceptionCode::VirtualInstruction) {
  if (TINST_VALUE_ON_VIRTUAL_INSTRUCTION == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VirtualInstruction exception");
  }
} else if (code == ExceptionCode::LoadAddressMisaligned) {
  if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadAddressMisaligned exception");
  }
} else if (code == ExceptionCode::LoadAccessFault) {
  if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadAccessFault exception");
  }
} else if (code == ExceptionCode::StoreAmoAddressMisaligned) {
  if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAddressMisaligned exception");
  }
} else if (code == ExceptionCode::StoreAmoAccessFault) {
  if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAccessFault exception");
  }
} else if (code == ExceptionCode::Ucall) {
  if (TINST_VALUE_ON_UCALL == "always zero") {
    return 0;
  } else {
```

```
    unpredictable("An unpredictable value is written into tinst in response to a UCall exception");
  }
} else if (code == ExceptionCode::Scall) {
  if (TINST_VALUE_ON_SCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a SCall exception");
  }
} else if (code == ExceptionCode::Mcall) {
  if (TINST_VALUE_ON_MCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a MCall exception");
  }
} else if (code == ExceptionCode::VScall) {
  if (TINST_VALUE_ON_VSCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VSCall exception");
  }
} else if (code == ExceptionCode::InstructionPageFault) {
  return 0;
} else if (code == ExceptionCode::LoadPageFault) {
  if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadPageFault exception");
  }
} else if (code == ExceptionCode::StoreAmoPageFault) {
  if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoPageFault exception");
  }
} else {
  assert(false, "Unhandled exception type");
}
```

D.167. gstage_page_walk

Translate guest physical address to physical address through a page walk.

May raise a Guest Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated physical address.

Return Type	`TranslationResult`
Arguments	`XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Boolean for_final_vs_pte, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadGuestPageFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionGuestPageFault : ExceptionCode::StoreAmoGuestPageFault);
Boolean mxr = for_final_vs_pte && (mstatus.MXR == 1);
Boolean pbmte = menvcfg.PBMTE == 1;
Boolean adue = menvcfg.ADUE == 1;
Bits<32> tinst = tinst_value_for_guest_page_fault(op, encoding, for_final_vs_pte);
```

```
U32 max_gpa_width = LEVELS * VPN_SIZE + 2 + 12;
if (gpaddr >> max_gpa_width != 0) {
  raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
}
ppn = hgatp.PPN;
for (U32 i = (LEVELS - 1); i >= 0; i--) {
  U32 this_vpn_size = (i == (LEVELS - 1)) ? VPN_SIZE + 2 : VPN_SIZE;
  U32 vpn = (gpaddr >> (12 + VPN_SIZE * i)) & 1 << this_vpn_size) - 1);   Bits<PA_SIZE> pte_paddr = (ppn << 12) + (vpn * (PTESIZE
/ 8;
  if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_paddr, PTESIZE)) {
    raise(access_fault_code, PrivilegeMode::U, vaddr);
  }
  access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
  Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_paddr);
  PteFlags pte_flags = pte[9:0];
  if ((VA_SIZE != 32) && (pte[58:54] != 0)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (!implemented?(ExtensionName::Svrsw60t59b)) {
    if ((PTESIZE >= 64) && pte[60:59] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
  }
  if (!implemented?(ExtensionName::Svnapot)) {
    if ((PTESIZE >= 64) && pte[63] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
  }
  if ((PTESIZE >= 64) && !pbmte && (pte[62:61] != 0)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if ((PTESIZE >= 64) && pbmte && (pte[62:61] == 3)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.V == 0) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.R == 0 && pte_flags.W == 1) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.R == 1 || pte_flags.X == 1) {
    if (pte_flags.U == 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    } else if ((op == MemoryOperation::Read) || (op == MemoryOperation::ReadModifyWrite)) {
      if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
      }
    }
    if ((i > 0) && (pte[(i - 1) * VPN_SIZE:0] != 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_paddr, PTESIZE)) {
          raise(access_fault_code, PrivilegeMode::U, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_paddr, PTESIZE)) {
          raise(access_fault_code, PrivilegeMode::U, vaddr);
        }
        access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
```

```
          success = atomic_check_then_write_64(pte_paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
          i = i + 1;
        } else {
          result.paddr = pte_paddr;
          if (PTESIZE >= 64) {
            result.pbmt = $enum(Pbmt, pte[62:61]);
          }
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
      }
    }
  } else {
    if (i == 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
  }
}
```

D.168. stage1_page_walk

Translate virtual address to physical address through a page walk.

May raise a Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated guest physical address.

Return Type	`TranslationResult`
Arguments	`Bits<MXLEN> vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadPageFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionPageFault : ExceptionCode::StoreAmoPageFault);
Boolean sse = false;
Boolean adue;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  adue = henvcfg.ADUE == 1;
} else {
  adue = menvcfg.ADUE == 1;
}
Boolean pbmte;
if (VA_SIZE == 32) {
  pbmte = false;
} else {
```

```
  if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    pbmte = henvcfg.PBMTE == 1;
  } else {
    pbmte = menvcfg.PBMTE == 1;
  }
}
Boolean mxr;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  mxr = (mstatus.MXR == 1) || (vsstatus.MXR == 1);
  ppn = vsatp.PPN;
} else {
  mxr = mstatus.MXR == 1;
  ppn = satp.PPN;
}
Boolean sum;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS)) {
  sum = vsstatus.SUM == 1;
} else {
  sum = mstatus.SUM == 1;
}
if ((VA_SIZE < xlen()) && (vaddr[xlen() - 1:VA_SIZE] != {xlen() - VA_SIZE{vaddr[VA_SIZE - 1]}})) {
  raise(page_fault_code, mode(), vaddr);
}
for (U32 i = (LEVELS - 1); i >= 0; i--) {
U32 vpn = (vaddr >> (12 + VPN_SIZE * i)) & 1 `<< VPN_SIZE) - 1);   Bits<PA_SIZE> pte_gpaddr = (ppn << 12) + (vpn * (PTESIZE / 8;
TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Read, effective_mode, encoding);
if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_phys.paddr, PTESIZE)) {
  raise(access_fault_code, mode(), vaddr);
}
access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_phys.paddr);
PteFlags pte_flags = pte[9:0];
Boolean ss_page = (pte_flags.R == 0) && (pte_flags.W == 1) && (pte_flags.X == 0);
if ((VA_SIZE != 32) && (pte[58:54] != 0)) {
  raise(page_fault_code, mode(), vaddr);
}
if (pte_flags.V == 0) {
  raise(page_fault_code, mode(), vaddr);
}
if (!sse) {
  if ((pte_flags.R == 0) && (pte_flags.W == 1)) {
    raise(page_fault_code, mode(), vaddr);
  }
}
if (pbmte) {
  if (pte[62:61] == 3) {
    raise(page_fault_code, mode(), vaddr);
  }
} else {
  if ((PTESIZE >= 64) && (pte[62:61] != 0)) {
    raise(page_fault_code, mode(), vaddr);
  }
}
if (!implemented?(ExtensionName::Svrsw60t59b)) {
  if ((PTESIZE >= 64) && pte[60:59] != 0) {
    raise(page_fault_code, mode(), vaddr);
  }
}
if (!implemented?(ExtensionName::Svnapot)) {
  if ((PTESIZE >= 64) && (pte[63] != 0)) {
    raise(page_fault_code, mode(), vaddr);
  }
}
if (pte_flags.R == 1 || pte_flags.X == 1) {
  if (op == MemoryOperation::Read || op == MemoryOperation::ReadModifyWrite) {
    if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
      raise(page_fault_code, mode(), vaddr);
    }
    if (effective_mode == PrivilegeMode::U && pte_flags.U == 0) {
      raise(page_fault_code, mode(), vaddr);
    } else if (misa.H == 1 && effective_mode == PrivilegeMode::VU && pte_flags.U == 0) {
      raise(page_fault_code, mode(), vaddr);
    } else if (effective_mode == PrivilegeMode::S && pte_flags.U == 1 && !sum) {
      raise(page_fault_code, mode(), vaddr);
    } else if (effective_mode == PrivilegeMode::VS && pte_flags.U == 1 && !sum) {
```

```
        raise(page_fault_code, mode(), vaddr);
      }
    }
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise(page_fault_code, mode(), vaddr);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise(page_fault_code, mode(), vaddr);
    } else if ((op == MemoryOperation::Fetch) && ss_page) {
      raise(page_fault_code, mode(), vaddr);
    }
    raise(page_fault_code, mode(), vaddr) if;
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Write, effective_mode, encoding);
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_phys.paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
          success = atomic_check_then_write_64(pte_phys.paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
          i = i + 1;
        } else {
          TranslationResult pte_phys = translate_gstage({pte[PA_SIZE - 3:i * VPN_SIZE + 10], vaddr[i * VPN_SIZE + 11:0]}, vaddr,
op, effective_mode, encoding);
          result.paddr = pte_phys.paddr;
          result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
        raise(page_fault_code, mode(), vaddr);
      }
    }
    TranslationResult pte_phys = translate_gstage({pte[PA_SIZE - 3:i * VPN_SIZE + 10], vaddr[i * VPN_SIZE + 11:0]}, vaddr, op,
effective_mode, encoding);
    result.paddr = pte_phys.paddr;
    if (PTESIZE >= 64) {
      result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
    }
    result.pte_flags = pte_flags;
    return result;
  } else {
    if (i == 0) {
      raise(page_fault_code, mode(), vaddr);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
      raise(page_fault_code, mode(), vaddr);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
      raise(page_fault_code, mode(), vaddr);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
      raise(page_fault_code, mode(), vaddr);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
  }
}
```

D.169. translate

Translate a virtual address for operation type op that appears to execute at effective_mode.

The translation will depend on the effective privilege mode.

May raise a Page Fault or Access Fault.

The final physical address is **not** access checked (for PMP, PMA, etc., violations). (though intermediate page table reads will be)

Return Type	`TranslationResult`
Arguments	`XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean cached_translation_valid;
CachedTranslationResult cached_translation_result;
cached_translation_result = cached_translation(vaddr, op);
if (cached_translation_result.valid) {
  return cached_translation_result.result;
}
TranslationResult result;
if (effective_mode == PrivilegeMode::M) {
  result.paddr = vaddr;
  return result;
}
SatpMode translation_mode = current_translation_mode(effective_mode);
if (translation_mode == SatpMode::Reserved) {
  if (op == MemoryOperation::Read) {
    raise(ExceptionCode::LoadPageFault, mode(), vaddr);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise(ExceptionCode::StoreAmoPageFault, mode(), vaddr);
  } else {
    assert(op == MemoryOperation::Fetch, "Unexpected memory operation");
    raise(ExceptionCode::InstructionPageFault, mode(), vaddr);
  }
}
if (translation_mode == SatpMode::Bare) {
  result.paddr = vaddr;
} else if (translation_mode == SatpMode::Sv32) {
  result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv39) {
  result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv48) {
  result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv57) {
  result = stage1_page_walk<57, 56, 64, 5>(vaddr, op, effective_mode, encoding);
} else {
  assert(false, "Unexpected SatpMode");
}
maybe_cache_translation(vaddr, op, result);
return result;
```

D.170. canonical_vaddr?

Returns whether or not *vaddr* is a valid (*i.e.*, canonical) virtual address.

If pointer masking (S**pm) is enabled, then vaddr will be masked before checking the canonical address.

Return Type	`Boolean`
Arguments	`XReg vaddr`

```
if (misa.S == 1'b0) {
  return true;
}
SatpMode satp_mode;
```

```
if (virtual_mode?()) {
  satp_mode = $enum(SatpMode, vsatp.MODE);
} else {
  satp_mode = $enum(SatpMode, satp.MODE);
}
XReg eaddr = mask_eaddr(vaddr);
if (SATP_MODE_BARE && (satp_mode == SatpMode::Bare)) {
  return true;
} else if ((MXLEN == 32) && satp_mode == SatpMode::Sv32) {
  return true;
} else if ((MXLEN == 64) && satp_mode == SatpMode::Sv39) {
  return eaddr[63:39] == {25{eaddr[38]}};
} else if ((MXLEN == 64) && satp_mode == SatpMode::Sv48) {
  return eaddr[63:48] == {16{eaddr[47]}};
} else if ((MXLEN == 64) && satp_mode == SatpMode::Sv57) {
  return eaddr[63:57] == {6{eaddr[56]}};
}
```

D.171. canonical_gpaddr?

Returns whether or not gpaddr is a valid (*i.e.*, canonical) guest physical address.

Return Type	Boolean
Arguments	XReg gpaddr

```
SatpMode satp_mode = $enum(SatpMode, satp.MODE);
if (satp_mode == SatpMode::Bare) {
  return true;
} else if (satp_mode == SatpMode::Sv32) {
  return true;
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv39)) {
  return gpaddr[63:39] == {25{gpaddr[38]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv48)) {
  return gpaddr[63:48] == {16{gpaddr[47]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv57)) {
  return gpaddr[63:57] == {6{gpaddr[56]}};
}
```

D.172. misaligned_is_atomic?

Returns true if an access starting at physical_address that is N bits long is atomic.

This function takes into account any Atomicity Granule PMAs, so **it should not be used for load-reserved/store-conditional**, since those PMAs do not apply to those accesses.

Return Type	Boolean
Arguments	Bits<PHYS_ADDR_WIDTH> physical_address

```
return false if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE == 0);
if (pma_applies?(PmaAttribute::MAG16, physical_address, N) && in_naturally_aligned_region?<128>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG8, physical_address, N) && in_naturally_aligned_region?<64>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG4, physical_address, N) && in_naturally_aligned_region?<32>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG2, physical_address, N) && in_naturally_aligned_region?<16>(physical_address, N)) {
  return true;
} else {
  return false;
}
```

D.173. read_memory_aligned

Read from virtual memory using a known aligned address.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault, effective_ldst_mode());
return read_physical_memory<LEN>(result.paddr);
```

D.174. read_memory

Read from virtual memory.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  return read_memory_aligned<LEN>(virtual_address, encoding);
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with large
atomicity granule");
  physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding).paddr :
virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
    return read_physical_memory<LEN>(physical_address);
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding).paddr :
virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::LoadAddressMisaligned, mode(), virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    Bits<LEN> result = 0;
    for (U32 I = 0; I < (LEN / 8); I++) {
      result = result | (read_memory_aligned<8>(virtual_address + I, encoding) `<< (8 * I));
    }
    return result;
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior when any part
of the misaligned access causes an exception");
  }
}
```

D.175. read_memory_xlen

Read XLEN bits from memory

Return Type	`Bits<MXLEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
if (xlen() == 32) {
  return read_memory<32>(virtual_address, encoding);
} else {
  return read_memory<64>(virtual_address, encoding);
}
```

D.176. write_memory_xlen

Read XLEN bits from memory

Return Type	`void`
Arguments	`XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
if (xlen() == 32) {
  return write_memory<32>(virtual_address, value, encoding);
} else {
  return write_memory<64>(virtual_address, value, encoding);
}
```

D.177. read_memory_xlen_aligned

Read from virtual memory XLEN (which may be runtime-determined) bits using a known aligned address.

Return Type	`Bits<MXLEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault, effective_ldst_mode());
if (xlen() == 32) {
  return read_physical_memory<32>(result.paddr);
} else {
  return read_physical_memory<64>(result.paddr);
}
```

D.178. invalidate_reservation_set

Invalidates any currently held reservation set.

This function may be called by the platform, independent of any actions occurring in the local hart, for any or no reason.

The platform **must** call this function if an external hart or device accesses part of this reservation set while reservation_set_valid could be true.

Return Type	`void`
Arguments	None

```
reservation_set_valid = false;
```

D.179. register_reservation_set

Register a reservation for a physical address range that subsumes [physical_address, physical_address + N).

Return Type	`void`
Arguments	`Bits<MXLEN> physical_address, Bits<MXLEN> length`

```
reservation_set_valid = true;
reservation_set_address = physical_address;
if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 64-byte region") {
  reservation_set_address = physical_address & ~MXLEN'h3f;
  reservation_set_size = 64;
} else if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 128-byte region") {
  reservation_set_address = physical_address & ~MXLEN'h7f;
  reservation_set_size = 128;
} else if (LRSC_RESERVATION_STRATEGY == "reserve exactly enough to cover the access") {
  reservation_set_address = physical_address;
  reservation_set_size = length;
} else if (LRSC_RESERVATION_STRATEGY == "custom") {
  unpredictable("Implementations may set reservation sets of any size, as long as they cover the reserved accessed");
} else {
  assert(false, "Unexpected LRSC_RESERVATION_STRATEGY");
}
```

D.180. load_reserved

Register a reservation for virtual_address at least N bits long and read the value from memory.

If aq is set, then also perform a memory model acquire.

If rl is set, then also perform a memory model release (software is discouraged from doing so).

This function assumes alignment checks have already occurred.

Return Type	`Bits<N>`
Arguments	`Bits<MXLEN> virtual_address, Bits<1>    aq, Bits<1>    rl, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::LoadAccessFault, mode(), virtual_address);
}
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
register_reservation_set(physical_address, N);
if (misa.S == 1 && LRSC_FAIL_ON_VA_SYNONYM) {
  reservation_virtual_address = virtual_address;
}
return read_memory_aligned<N>(physical_address, encoding);
```

D.181. store_conditional

Atomically check the reservation set to ensure:

- it is valid

- it covers the region addressed by this store
- the address setting the reservation set matches virtual address

If the preceding are met, perform the store and return 0. Otherwise, return 1.

Return Type	Boolean
Arguments	Bits<MXLEN> virtual_address, Bits<MXLEN> value, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
}
access_check(physical_address, N, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
if (reservation_set_valid == false) {
  return false;
}
if (!contains?(reservation_set_address, reservation_set_size, physical_address, N)) {
  invalidate_reservation_set();
  return false;
}
if (LRSC_FAIL_ON_NON_EXACT_LRSC) {
  if (reservation_physical_address != physical_address || reservation_size != N) {
    invalidate_reservation_set();
    return false;
  }
}
if (LRSC_FAIL_ON_VA_SYNONYM) {
  if (reservation_virtual_address != virtual_address || reservation_size != N) {
    invalidate_reservation_set();
    return false;
  }
}
write_physical_memory<N>(physical_address, value);
return true;
```

D.182. amo

Atomically read-modify-write the location at virtual_address.

The value written to virtual_address will depend on op.

If aq is 1, then the amo also acts as a memory model acquire. If rl is 1, then the amo also acts as a memory model release.

Return Type	Bits<N>
Arguments	XReg virtual_address, Bits<N> value, AmoOperation op, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<N>(virtual_address);
if (!aligned && MISALIGNED_LDST_EXCEPTION_PRIORITY == "high") {
  raise(ExceptionCode::StoreAmoAddressMisaligned, mode(), virtual_address);
}
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::ReadModifyWrite,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::AmoNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
} else if (op == AmoOperation::Add || op == AmoOperation::Max || op == AmoOperation::Maxu || op == AmoOperation::Min || op ==
```

```
AmoOperation::Minu && !pma_applies?(PmaAttribute::AmoArithmetic, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
} else if (op == AmoOperation::And || op == AmoOperation::Or || op == AmoOperation::Xor && !pma_applies?(PmaAttribute::AmoLogical,
physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
} else {
  assert(pma_applies?(PmaAttribute::AmoSwap, physical_address, N) && op == AmoOperation::Swap, "Bad AMO operation");
}
if (!aligned && !misaligned_is_atomic?<N>(physical_address)) {
  raise(ExceptionCode::StoreAmoAddressMisaligned, mode(), virtual_address);
}
if (N == 32) {
  return atomic_read_modify_write_32(physical_address, value, op);
} else {
  return atomic_read_modify_write_64(physical_address, value, op);
}
```

D.183. write_memory_aligned

Write to virtual memory using a known aligned address.

Return Type	`void`
Arguments	`XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
write_physical_memory<LEN>(physical_address, value);
```

D.184. write_memory

Write to virtual memory

Return Type	`void`
Arguments	`XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  write_memory_aligned<LEN>(virtual_address, value, encoding);
  return ;
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with large
atomicity granule");
  physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
    write_physical_memory<LEN>(physical_address, value);
    return ;
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::StoreAmoAddressMisaligned, mode(), virtual_address);
```

```
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    for (U32 I = 0; I < (LEN / 8); I++) {
      write_memory_aligned<8>(virtual_address + I, (value >> (8 * I))[7:0], encoding);
    }
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior when any part
of the misaligned access causes an exception");
  }
}
```

D.185. write_memory_xlen_aligned

Write to virtual memory XLEN bits (which may be runtime determined) using a known aligned address.

Return Type	`void`
Arguments	`XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
access_check(physical_address, xlen(), virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
  write_physical_memory<32>(physical_address, value);
} else {
  write_physical_memory<64>(physical_address, value);
}
```

D.186. mstatus_sd_has_known_reset

Returns true if the mstatus.SD bit has a defined reset value, as determined by various parameters.

Return Type	`Boolean`
Arguments	None

```
Boolean fs_has_single_value = !implemented?(ExtensionName::F || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1));
Boolean vs_has_single_value = !implemented?(ExtensionName::V || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1));
return fs_has_single_value && vs_has_single_value;
```

D.187. mstatus_sd_reset_value

Returns the reset value of mstatus.SD when known

Return Type	`Bits①`
Arguments	None

```
assert(mstatus_sd_has_known_reset(), "mstatus_sd_reset_value is only defined when mstatus_sd_has_known_reset() == true");
Bits<2> fs_value, vs_value;
if ((!implemented?(ExtensionName::F)) || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1)) {
  fs_value = (!implemented?(ExtensionName::F)) ? 0 : MSTATUS_FS_LEGAL_VALUES[0];
}
if ((!implemented?(ExtensionName::V)) || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1)) {
  fs_value = (!implemented?(ExtensionName::V)) ? 0 : MSTATUS_VS_LEGAL_VALUES[0];
}
return fs_value == 3) || (vs_value == 3 ? 1 : 0;
```

D.188. check_zcmt_enabled

If the Smstateen extension is implemented, then bit 2 in mstateen0, sstateen0, and hstateen0 is implemented. If bit 2 of a controlling `stateen0` CSR is zero, then access to the jvt CSR and execution of a cm.jalt or cm.jt instruction by a lower privilege level results in an illegal-instruction trap (or, if appropriate, a virtual-instruction trap).

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

```
if ((mode() != PrivilegeMode::M && implemented?(ExtensionName::Smstateen) && mstateen0.JVT == 1'b0) || (mode() == PrivilegeMode::U
&& implemented?(ExtensionName::Ssstateen) && sstateen0.JVT == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else if ((mode() == PrivilegeMode::VS && implemented?(ExtensionName::Ssstateen) && hstateen0.JVT == 1'b0) || (mode() ==
PrivilegeMode::VU && implemented?(ExtensionName::Ssstateen) && (sstateen0.JVT == 1'b0 || hstateen0.JVT == 1'b0))) {
  raise(ExceptionCode::VirtualInstruction, mode(), encoding);
}
```