MC100-32 Processor Certification Requirements Document

Table of Contents

CRD Revision History

History of documentation changes that eventually lead to releases.

Date	Revision	Changes
2025-10-29	0.9.0	• Made Zicntr from mandatory to optional since recent published ISA manuals have it as "recommended"
2025-01-19	0.8.0	• Updated so that content can apply equally to all certificate-related documents such as CRDs (Certification Requirement Documents) and CTPs (Certification Test Plans).
2024-07-29	0.7.0	• First version after moving non-microcontroller content in this document to a new document called "RISC-V CRDs (Certification Requirement Documents)" • Change MC100 Unpriv ISA spec from "riscv-spec-v2.1, May 31, 2016" to github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC since the former isn't ratified by the latter is the oldest ratified version. • Added requirements for WFI instruction • Added requirements related to msip memory-mapped register
2024-07-11	0.6.0	• Supporting multiple MC versions to support customers wanting to certify existing microcontrollers not using the latest version of ratified standards. • Changed versioning scheme to use major.minor.patch instead of 3-digit major & minor. • Added a table showing the mapping from MC version to ISA manuals. • Reluctantly made interrupts OUT OF SCOPE for MC100 since only the CLINT interrupt controller was ratified at that time and isn't anticipated to be the interrupt controller used by MC100 implementations. • Clarified MANDATORY behaviors for mie and mip CSRs • Removed canonical discovery recipe because the OPT-* options directly inform the certification tests and certification reference model of the status of the various options. Also, canonical discovery recipes (e.g., probing for CLIC) violate the certification approach of avoiding writing potentially illegal values to CSR fields. • Added more options for interrupts • Moved non-microcontroller content in this document to a new document called "RISC-V Certification Plans"
2024-06-03	0.5.0	• Renamed to "RISC-V Microcontroller Certification Plan" based on Jason's recommendation • Added mvendorid, marchid, mimpid, and mhardid read-only priv CSRs because Allen pointed out these are mandatory in M-mode v1.13 (probably older versions too, haven't looked yet). • Added table showing mapping of MC versions to associated RISC-V specifications

Date	Revision	Changes
2024-06-03	0.4.0	• Added M-mode instruction requirements • Made Zicntr MANDATORY due to very low cost for implementations to support (in the spirit of minimizing options). • Removed OPT-CNTR-PREC since minstret and mcycle must be a full 64 bits to be standard-compliant.
2024-05-25	0.3.0	• Includes Zicntr as OPTIONAL and then has only 32-bit counters for instret and cycle.
2024-05-20	0.2.0	• Very early draft
2024-05-16	0.1.0	• Initial version

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

Term	Meaning
Should	Indicates a recommended requirement
May	Indicates an optional requirement

1. Introduction

The MC100 Processor Certificate targets basic RISC-V microcontrollers. It supports either a 32-bit (MC100-32) or 64-bit (MC100-64) base ISA. MC100 is not intended for the smallest possible microcontrollers but rather for applications benefiting from a minimal but standardized microcontroller. It consists of:

- Unprivileged ISA: RV32I for MC100-32 and RV64I for MC100-64 with a few extensions suitable for a basic microcontroller.
- Privileged ISA: Only the M-mode features listed as mandatory in the RISC-V Privileged ISA manual

The MC (Microcontroller Class) targets processors running low-level software on an RTOS or bare-metal.

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
MC100-32	No profile	20191213	20190608-Priv-MSU-Ratified	0.13.2

1.5. Privileged Modes

M	S	U	HS	VS	VU
IN-SCOPE	OUT-OF-SCOPE	OUT-OF-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 MC100-32 certificate doesn't cover their behaviors.

2.1. Mandatory Extensions

The MC100-32 certificate has 4 mandatory extensions.

Requirement ID	Extension	Version	Long Name	Note
REQ-EXT-C	C	~> 2.2	Compressed instructions	
REQ-EXT-I	I	~> 2.1	Base integer ISA (RV32I or RV64I)	
REQ-EXT-Sm	Sm	~> 1.11.0	Machine mode	
REQ-EXT-Zicsr	Zicsr	~> 2.0	Control and status register instructions	

2.2. Optional Extensions

The MC100-32 certificate has 2 optional extensions.

Requirement ID	Extension	Version	Long Name	Note
REQ-EXT-M	M	~> 2.0	Integer multiply and divide	
REQ-EXT-Zicntr	Zicntr	~> 2.0	Base Counters and Timers	

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.

The MC100-32 certificate has 21 IN-SCOPE parameters.

Parameter	Type	Allowed Value(s)	Extension(s)	Note
ARCH_ID_VALUE	32-bit integer, ≠ 0 or 0x80000000	Any		
IMP_ID_VALUE	32-bit integer	Any		
MISALIGNED_LDST	boolean	Any		
MISALIGNED_LDST_EXCEPTION_PRIORITY	[low, high]	Any		
MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE	[0, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]	Any		
MISALIGNED_SPLIT_STRATEGY	[sequential_bytes, custom]	sequential_bytes		
MISA_CSR_IMPLEMENTED	boolean	Any		
MTVAL_WIDTH	0 to 32	Any		
MTVEC_BASE_ALIGNMENT_DIRECT	[4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4095, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864, 134217728, 268435456, 536870912, 1073741824, 2147483648]	Any		
MTVEC_BASE_ALIGNMENT_VECTORED	[4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4095, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608, 16777216, 33554432, 67108864, 134217728, 268435456, 536870912, 1073741824, 2147483648]	Any		
MTVEC_MODES	1-element to 2-element array of [0, 1]	Any		
MXLEN	[32, 64]	32		
MXLEN	[32, 64]	32		
M_MODE_ENDIANNESS	[little, big, dynamic]	little		
PHYS_ADDR_WIDTH	1 to 34	Any		
PRECISE_SYNCHRONOUS_EXCEPTIONS	boolean	true		
TIME_CSR_IMPLEMENTED	boolean	Any		
TRAP_ON_EBREAK	boolean	true		
TRAP_ON_ECALL_FROM_M	boolean	true		
VENDOR_ID_BANK	25-bit integer	Any		
VENDOR_ID_OFFSET	7-bit integer	Any		

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 MC100-32 certificate has 18 OUT-OF-SCOPE parameters.

Parameters	Type	Extension(s)
COUNTINHIBIT_EN	32-element array where: [0] is boolean [1] is false [2] is boolean additional items are: boolean	
MARCHID_IMPLEMENTED	boolean	
MIMPID_IMPLEMENTED	boolean	
MTVEC_ACCESS	[ro, rw]	
MTVEC_ILLEGAL_WRITE_BEHAVIOR	[retain, custom]	
PMA_GRANULARITY	2 to 66	
REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION	boolean	
REPORT_VA_IN_MTVAL_ON_BREAKPOINT	boolean	
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT	boolean	
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED	boolean	
REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT	boolean	
REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED	boolean	
REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT	boolean	
REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED	boolean	
TRAP_ON_ILLEGAL_WLRL	boolean	
TRAP_ON_RESERVED_INSTRUCTION	boolean	
TRAP_ON_UNIMPLEMENTED_CSR	boolean	
TRAP_ON_UNIMPLEMENTED_INSTRUCTION	boolean	

4. Instruction Summary

The MC100-32 certificate has up to 59 instructions (exact number depends on an implementation's options).

Name	Long Name
add	Integer add
addi	Add immediate
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
csrrc	Atomic Read and Clear Bits in CSR
csrrci	Atomic Read and Clear Bits in CSR with Immediate
csrrs	Atomic Read and Set Bits in CSR
csrrsi	Atomic Read and Set Bits in CSR with Immediate
csrrw	Atomic Read/Write CSR
csrrwi	Atomic Read/Write CSR Immediate
div	Signed division
divu	Unsigned division
ebreak	Breakpoint exception
ecall	Environment call

Name	Long Name
fence	Memory ordering fence
fence.tso	Memory ordering fence, total store ordering
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
mret	Machine-mode Return from Trap
mul	Signed multiply
mulh	Signed multiply high
mulhsu	Signed/unsigned multiply high
mulhu	Unsigned multiply high
or	Or
ori	Or immediate
rem	Signed remainder
remu	Unsigned remainder
sb	Store byte
sd	Store doubleword
sh	Store halfword
sll	Shift left logical
slli	Shift left logical immediate
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
srl	Shift right logical
srli	Shift right logical immediate
sub	Subtract
sw	Store word
wfi	Wait for interrupt
xor	Exclusive Or
xori	Exclusive Or immediate

5. CSR Summary

The MC100-32 certificate has up to 19 CSRs (exact number depends on an implementation's options).

5.1. By Name

Name	Long Name	Address	Mode	Defining Extension(s)
cycle	Cycle counter for RDCYCLE Instruction	0xc00	U	Zicntr any
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U	Zicntr any
marchid	Machine Architecture ID	0xf12	M	Sm any
mcause	Machine Cause	0x342	M	Sm any
mcountinhibit	Machine Counter Inhibit	0x320	M	Sm any

Name	Long Name	Address	Mode	Defining Extension(s)
mcycle	Machine Cycle Counter	0xb00	M	Sm any
mepc	Machine Exception Program Counter	0x341	M	Sm any
mhartid	Machine Hart ID	0xf14	M	Sm any
mie	Machine Interrupt Enable	0x304	M	Sm any
mimpid	Machine Implementation ID	0xf13	M	Sm any
minstret	Machine Instructions Retired Counter	0xb02	M	Sm any
mip	Machine Interrupt Pending	0x344	M	Sm any
misa	Machine ISA Control	0x301	M	Sm any
mscratch	Machine Scratch Register	0x340	M	Sm any
mstatus	Machine Status	0x300	M	Sm any
mtval	Machine Trap Value	0x343	M	Sm any
mtvec	Machine Trap Vector Control	0x305	M	Sm any
mvendorid	Machine Vendor ID	0xf11	M	Sm any
time	Timer for RDTIME Instruction	0xc01	U	Zicntr any

5.2. By Address

Address	Mode	Name	Long Name	Primary Extension
0x300	M	mstatus	Machine Status	Sm any
0x301	M	misa	Machine ISA Control	Sm any
0x304	M	mie	Machine Interrupt Enable	Sm any
0x305	M	mtvec	Machine Trap Vector Control	Sm any
0x320	M	mcountinhibit	Machine Counter Inhibit	Sm any
0x340	M	mscratch	Machine Scratch Register	Sm any
0x341	M	mepc	Machine Exception Program Counter	Sm any
0x342	M	mcause	Machine Cause	Sm any
0x343	M	mtval	Machine Trap Value	Sm any
0x344	M	mip	Machine Interrupt Pending	Sm any
0xb00	M	mcycle	Machine Cycle Counter	Sm any
0xb02	M	minstret	Machine Instructions Retired Counter	Sm any
0xc00	U	cycle	Cycle counter for RDCYCLE Instruction	Zicntr any
0xc01	U	time	Timer for RDTIME Instruction	Zicntr any
0xc02	U	instret	Instructions retired counter for RDINSTRET Instruction	Zicntr any
0xf11	M	mvendorid	Machine Vendor ID	Sm any
0xf12	M	marchid	Machine Architecture ID	Sm any
0xf13	M	mimpid	Machine Implementation ID	Sm any
0xf14	M	mhartid	Machine Hart ID	Sm any

Appendix A: Extension Details

A.1. Extension C

Long Name: Compressed instructions
Version Requirement: ~> 2.2

A.1.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-12

A.1.2. 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.1.3. 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.1.4. 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. Extension I

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

A.2.1. Available Versions

	Version 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.2.2. Synopsis

Base integer instructions — TODO

A.2.3. Instructions

The following 43 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**
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**
or	**Or**
ori	**Or immediate**
sb	**Store byte**
sd	**Store doubleword**
sh	**Store halfword**
sll	**Shift left logical**
slli	**Shift left logical immediate**
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**
srl	**Shift right logical**
srli	**Shift right logical immediate**
sub	**Subtract**
sw	**Store word**
xor	**Exclusive Or**
xori	**Exclusive Or immediate**

A.3. Extension M

Long Name: Integer multiply and divide
Version Requirement: ~> 2.0

A.3.1. Available Versions

Version 2.0.0	
State	ratified
Ratification date	2019-12

A.3.2. 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.3.3. Instructions

The following 8 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**
mul	**Signed multiply**

mulh	**Signed multiply high**
mulhsu	**Signed/unsigned multiply high**
mulhu	**Unsigned multiply high**
rem	**Signed remainder**
remu	**Unsigned remainder**

A.4. Extension Sm

Long Name: Machine mode
Version Requirement: ~> 1.11.0

A.4.1. Available Versions

	Version 1.11.0
State	ratified
Ratification date	2019-12
Changes	• Moved Machine spec to **Ratified** status. • Improvements to the description and commentary. • Specified which interrupt sources are reserved for standard use. • Allocated some synchronous exception causes for custom use. • Specified the priority ordering of synchronous exceptions. • Added specification that xRET instructions may, but are not required to, clear LR reservations if A extension present. • Made the mstatus.MPP field **WARL**, rather than **WLRL**. • Made the unused `xip` fields **WPRI**, rather than **WIRI**. • Made the unused misa fields **WARL**, rather than **WIRI**. • Rectified an editing error that misdescribed the mechanism by which mstatus is written upon an exception. • Described scheme for emulating misaligned AMOs. • Specified the behavior of the misa and `xepc` registers in systems with variable IALIGN. • Specified the behavior of writing self-contradictory values to the misa register. • Specified contents of CSRs across XLEN modification. • Moved PLIC chapter into its own document.

	Version 1.12.0
State	ratified
Ratification date	2021-12

Version 1.12.0	
Changes	• Changed MRET to clear mstatus.MPRV when leaving M-mode. • Relaxed I/O regions have been specified to follow RVWMO. The previous specification implied that PPO rules other than fences and acquire/release annotations did not apply. • Constrained the LR/SC reservation set size and shape when using page-based virtual memory. • PMP changes require an SFENCE.VMA on any hart that implements page-based virtual memory, even if VM is not currently enabled. • Removed the N extension. • Defined the mandatory RV32-only CSR mstatush, which contains most of the same fields as the upper 32 bits of RV64's mstatus. • Defined the mandatory CSR mconfigptr, which if nonzero contains the address of a configuration data structure. • Defined mseccfg and mseccfgh CSRs, which control the machine's security configuration. • Defined menvcfg CSR (and RV32-only menvcfgh), which control various characteristics of the execution environment. • Designated part of SYSTEM major opcode for custom use. • Permitted the unconditional delegation of less-privileged interrupts. • Added optional big-endian and bi-endian support. • Made priority of load/store/AMO address-misaligned exceptions implementation-defined relative to load/store/AMO page-fault and access-fault exceptions. • Software breakpoint exceptions are permitted to write either 0 or the `pc` to `xtval`. • Specified relaxed constraints for implicit reads of non-idempotent regions.

Version 1.13.0	
State	frozen
Changes	• Redefined misa.MXL to be read-only, making MXLEN a constant. • Defined the misa.B field to reflect that the B extension has been implemented. • Defined the misa.V field to reflect that the V extension has been implemented. • Defined the RV32-only medelegh CSR. • Defined the misaligned atomicity granule PMA, superseding the proposed Zam extension. • Defined hardware error and software check exception codes. • Specified synchronization requirements when changing the PBMTE fields in menvcfg and henvcfg. • Exposed count-overflow interrupts to VS-mode via the Shlcofideleg extension. • Relaxed behavior of some HINTs when MXLEN > XLEN. • Transliterated the document from LaTeX into AsciiDoc. • Included all ratified extensions through March 2024. • Clarified that "platform- or custom-use" interrupts are actually "platform-use interrupts", where the platform can choose to make some custom. • Clarified semantics of explicit accesses to CSRs wider than XLEN bits. • Clarified that MXLEN≥SXLEN. • Clarified that WFI is not a HINT instruction. • Clarified that, for a given exception cause, `xtval` might sometimes be set to a nonzero value but sometimes not. • Clarified exception behavior of unimplemented or inaccessible CSRs. • Replaced the concept of vacant memory regions with inaccessible memory or I/O regions. • Clarified that timer and count-overflow interrupts' arrival in interrupt-pending registers is not immediate. • Clarified that MXR affects only explicit memory accesses.

A.4.2. Synopsis

This chapter describes the machine-level operations available in machine-mode (M-mode), which is the highest privilege mode in a RISC-V hart. M-mode is used for low-level access to a hardware platform and is the first mode entered at reset. M-mode can also be used to implement features that are too difficult or expensive to implement in hardware directly. The RISC-V machine-level ISA contains a common core that is extended depending on which other privilege levels are supported and other details of the hardware implementation. This chapter describes the RISC-V machine-level architecture, which contains a common core that is used with various supervisor-level address translation and protection schemes.

A.4.3. Instructions

The following 2 instructions are added by extension version 1.11.0 (the minimum version of this extension that satifies the extension requirement).

mret	**Machine-mode Return from Trap**
wfi	**Wait for interrupt**

A.4.4. CSRs

The following 16 CSRs are added by extension version 1.11.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
marchid	Machine Architecture ID	0xf12	M
mcause	Machine Cause	0x342	M
mcountinhibit	Machine Counter Inhibit	0x320	M
mcycle	Machine Cycle Counter	0xb00	M
mepc	Machine Exception Program Counter	0x341	M
mhartid	Machine Hart ID	0xf14	M
mie	Machine Interrupt Enable	0x304	M
mimpid	Machine Implementation ID	0xf13	M
minstret	Machine Instructions Retired Counter	0xb02	M
mip	Machine Interrupt Pending	0x344	M
misa	Machine ISA Control	0x301	M
mscratch	Machine Scratch Register	0x340	M
mstatus	Machine Status	0x300	M
mtval	Machine Trap Value	0x343	M
mtvec	Machine Trap Vector Control	0x305	M
mvendorid	Machine Vendor ID	0xf11	M

A.4.5. OUT-OF-SCOPE Parameters

COUNTINHIBIT_EN ⇒ 32-element array where:
  [0] is boolean
  [1] is false
  [2] is boolean
additional items are:
  boolean::

+

Indicates which hardware performance monitor counters can be disabled from mcountinhibit.

An unimplemented counter cannot be specified, i.e., if HPM_COUNTER_EN[3] is false, it would be illegal to set COUNTINHIBIT_EN[3] to true.

COUNTINHIBIT_EN[1] can never be true, since it corresponds to mcountinhibit, which is always read-only-0.

MARCHID_IMPLEMENTED ⇒ boolean

- false: marchid is not implemented, and must be read-only-0
- true: marchid is implemented, and the value is determined by `ARCH_ID_VALUE`

MIMPID_IMPLEMENTED ⇒ boolean

- false: mimpid is not implemented, and must be read-only-0
- true: mimpid is implemented, and the value is determined by `IMP_ID_VALUE`

MTVEC_ACCESS ⇒ [ro, rw]

Options:

ro

mtvec is read-only.

rw

mtvec is read-write, but may not accept all values.

MTVEC_ILLEGAL_WRITE_BEHAVIOR ⇒ [retain, custom]

Options:

retain

When either mtvec.MODE or mtvec.BASE is illegal, mtvec will retain its current value

custom

When either mtvec.MODE or mtvec.BASE is illegal, mtvec will obtain an unpredictable value

Other values may be added over time once other common behaviors are identified.

PMA_GRANULARITY ⇒ 2 to 66

Generally, for systems with an MMU, should not be smaller than 12, as that would preclude caching PMA results in the TLB along with virtual memory translations

REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ⇒ boolean

Options:

- true: mtval is written with the encoding of an instruction causing an IllegalInstruction exception
- false: mtval is written with 0 when an instruction causes an IllegalInstruction exception.

REPORT_VA_IN_MTVAL_ON_BREAKPOINT ⇒ boolean

Options:

- true: mtval is written with the virtual PC of an EBREAK instruction (same information as mepc).
- false: mtval is written with 0 on an EBREAK instruction.

Regardless, mtval is always written with a virtual PC when an external breakpoint is generated

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ⇒ boolean

Options:

- true: mtval is written with the virtual address of a fetch causing the access fault
- false: mtval is written with 0 when a fetch causes an access fault

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ⇒ boolean

Options:

- true: mtval is written with the virtual address of a trapping misaligned fetch
- false: mtval is written with 0 when a misaligned fetch traps

REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ⇒ boolean

Options:

- true: mtval is written with the virtual address of a load causing the access fault
- false: mtval is written with 0 when a load causes an access fault

REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ⇒ boolean

Options:

- true: mtval is written with the virtual address of a trapping misaligned load.
- false: mtval is written with 0 when a misaligned load traps.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ⇒ boolean

Options:

- true: mtval is written with the virtual address of a store or AMO causing the access fault
- false: mtval is written with 0 when a store or AMO causes an access fault

REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ⇒ boolean

Options:

- true: mtval is written with the virtual address of a trapping misaligned store or AMO.
- false: mtval is written with 0 when a misaligned store or AMO traps.

TRAP_ON_ILLEGAL_WLRL ⇒ boolean

Options:

- true: Writing an illegal value to a WLRL CSR field will cause an IllegalInstruction exception.

- false: Writing an illegal value to a WLRL CSR field causes unpredictable behavior.

TRAP_ON_RESERVED_INSTRUCTION ⇒ boolean

Options:

- true: Fetching an unimplemented and/or undefined instruction from the standard/reserved opcode space will cause an IllegalInstruction exception.
- false: Fetching an unimplemented and/or undefined instruction from the standard/reserved opcose space causes unpredictable behavior.

TRAP_ON_RESERVED_INSTRUCTION may be false while TRAP_ON_UNIMPLEMENTED_INSTRUCTION is true when a custom instruction is implemented in the standard/reserved opcode space.

TRAP_ON_UNIMPLEMENTED_CSR ⇒ boolean

Options:

- true: Accessing an unimplemented CSR (via a Zicsr instruction) will cause an IllegalInstruction exception.
- false: Accessing an unimplemented CSR (via a Zicsr instruction) will cause unpredictable behavior.

TRAP_ON_UNIMPLEMENTED_INSTRUCTION ⇒ boolean

Options:

- true: Fetching an unimplemented instruction will cause an IllegalInstruction exception.
- false: Fetching an unimplemented instruction causes unpredictable behavior.

An unimplemented instruction is any instruction encoding that is not defined by the implementation. Custom instructions are considered implemented.

A.5. Extension Zicntr

Long Name: Base Counters and Timers
Version Requirement: ~> 2.0

A.5.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-12

A.5.2. Synopsis

The CYCLE, TIME, and INSTRET counters, which have dedicated functions (cycle count, real-time clock, and instructions retired, respectively).

A.5.3. CSRs

The following 3 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
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U
time	Timer for RDTIME Instruction	0xc01	U

A.5.4. Parameters

The following parameters (implementation options) may affect the operation of this extension:

TIME_CSR_IMPLEMENTED

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.6. Extension Zicsr

Long Name: Control and status register instructions
Version Requirement: ~> 2.0

A.6.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-04

A.6.2. Synopsis

Control and status register instructions

A.6.3. Instructions

The following 6 instructions are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

csrrc	**Atomic Read and Clear Bits in CSR**
csrrci	**Atomic Read and Clear Bits in CSR with Immediate**
csrrs	**Atomic Read and Set Bits in CSR**
csrrsi	**Atomic Read and Set Bits in CSR with Immediate**
csrrw	**Atomic Read/Write CSR**
csrrwi	**Atomic Read/Write CSR Immediate**

Appendix B: Instruction Details

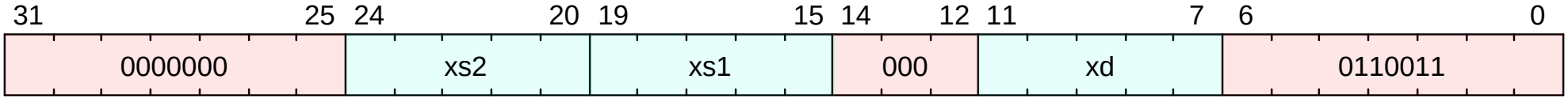
B.1. add

Integer add

This instruction is defined by:

I

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.

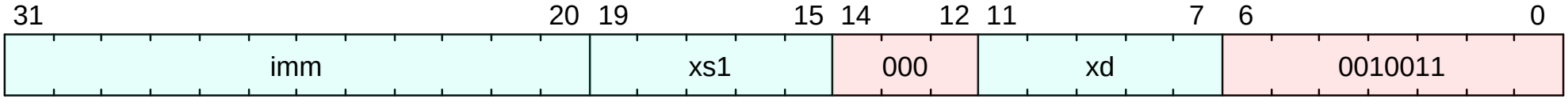
B.2. addi

Add immediate

This instruction is defined by:

I

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.

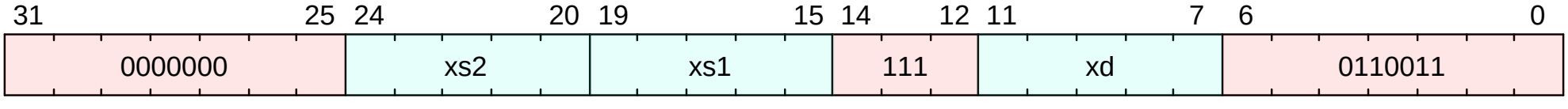
B.3. and

And

This instruction is defined by:

I

B.3.1. Encoding



B.3.2. Description

And xs1 with xs2, and store the result in xd

B.3.3. Access

M
Always

B.3.4. Decode Variables

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

B.3.5. IDL Operation

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

B.3.6. 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.3.7. Exceptions

This instruction does not generate synchronous exceptions.

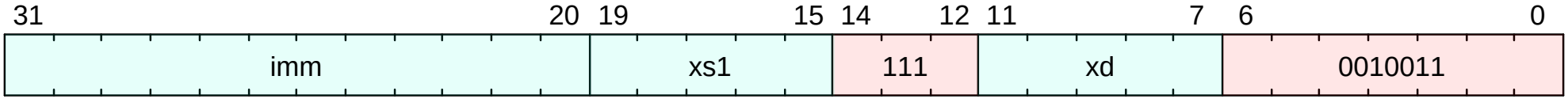
B.4. andi

And immediate

This instruction is defined by:

I

B.4.1. Encoding



B.4.2. Description

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

B.4.3. Access

M
Always

B.4.4. Decode Variables

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

B.4.5. IDL Operation

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

B.4.6. 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.4.7. Exceptions

This instruction does not generate synchronous exceptions.

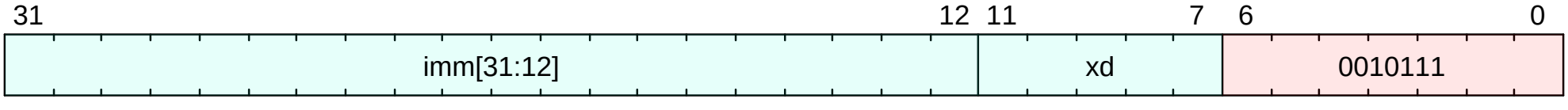
B.5. auipc

Add upper immediate to pc

This instruction is defined by:

I

B.5.1. Encoding



B.5.2. Description

Add an immediate to the current PC.

B.5.3. Access

M
Always

B.5.4. Decode Variables

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

B.5.5. IDL Operation

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

B.5.6. 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.5.7. Exceptions

This instruction does not generate synchronous exceptions.

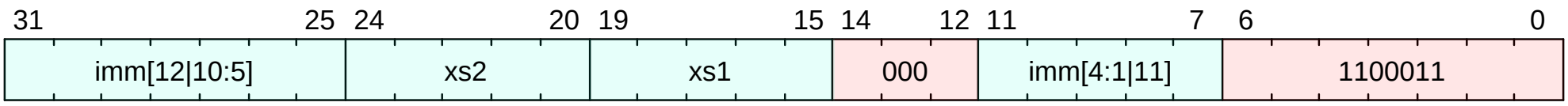
B.6. beq

Branch if equal

This instruction is defined by:

I

B.6.1. Encoding



B.6.2. 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.6.3. Access

M
Always

B.6.4. Decode Variables

```
signed Bits<13> imm = sext({$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.6.5. IDL Operation

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

B.6.6. 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.6.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

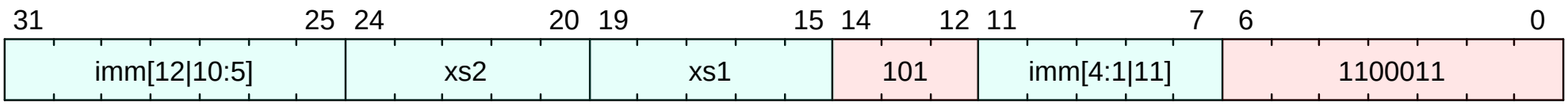
B.7. bge

Branch if greater than or equal

This instruction is defined by:

I

B.7.1. Encoding



B.7.2. 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.7.3. Access

M
Always

B.7.4. Decode Variables

```
signed Bits<13> imm = sext({$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.7.5. IDL Operation

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

B.7.6. 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.7.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

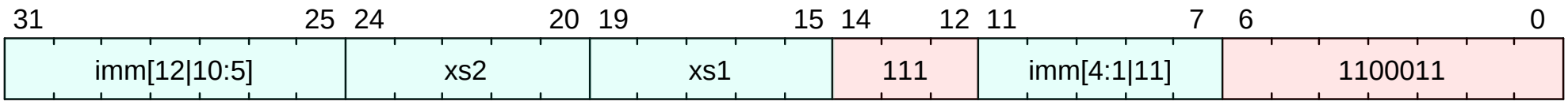
B.8. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

I

B.8.1. Encoding



B.8.2. 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.8.3. Access

M
Always

B.8.4. 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.8.5. IDL Operation

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

B.8.6. 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.8.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

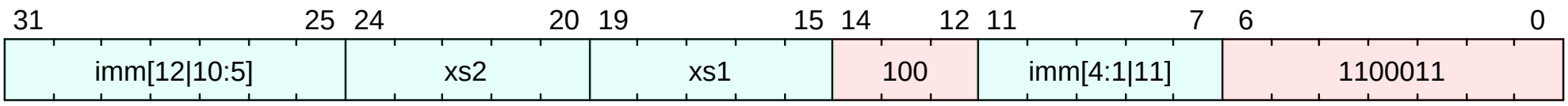
B.9. blt

Branch if less than

This instruction is defined by:

I

B.9.1. Encoding



B.9.2. 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.9.3. Access

M
Always

B.9.4. Decode Variables

```
signed Bits<13> imm = sext({$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.9.5. IDL Operation

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

B.9.6. 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.9.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

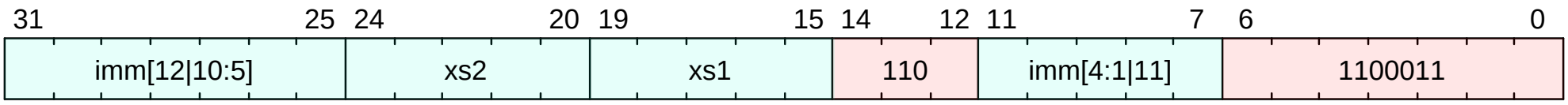
B.10. bltu

Branch if less than unsigned

This instruction is defined by:

I

B.10.1. Encoding



B.10.2. 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.10.3. Access

M
Always

B.10.4. 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.10.5. IDL Operation

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

B.10.6. 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.10.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

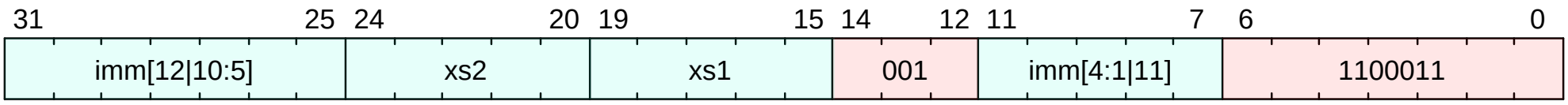
B.11. bne

Branch if not equal

This instruction is defined by:

I

B.11.1. Encoding



B.11.2. 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.11.3. Access

M
Always

B.11.4. Decode Variables

```
signed Bits<13> imm = sext({$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.11.5. IDL Operation

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

B.11.6. 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.11.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

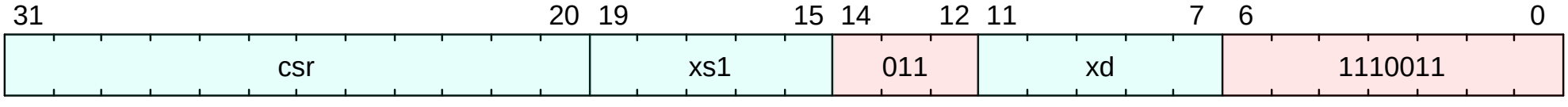
B.12. csrrc

Atomic Read and Clear Bits in CSR

This instruction is defined by:

Zicsr

B.12.1. Encoding



B.12.2. Description

The CSRRC (Atomic Read and Clear Bits in CSR) instruction reads the value of the CSR, zero-extends the value to XLEN bits, and writes it to integer register `xd`. The initial value in integer register `xs1` is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in `xs1` will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable.

For CSRRC, if `xs1=x0`, then the instruction will not write to the CSR at all, and so shall not cause any of the side effects that might otherwise occur on a CSR write, nor raise illegal- instruction exceptions on accesses to read-only CSRs. CSRRC always reads the addressed CSR and cause any read side effects regardless of `xs1` and `xd` fields. Note that if `xs1` specifies a register other than `x0`, and that register holds a zero value, the instruction will not action any attendant per-field side effects, but will action any side effects caused by writing to the entire CSR.

B.12.3. Access

M
Always

B.12.4. Decode Variables

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

B.12.5. IDL Operation

```
Csr csr_handle = direct_csr_lookup(csr);
Boolean will_write = xs1 != 0;
if (csr_handle.valid == false) {
  unimplemented_csr($encoding);
} else if (!compatible_mode?(csr_handle.mode, mode())) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if (will_write && csr_handle.writable == false) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg initial_csr_value = csr_sw_read(csr_handle);
if (xs1 != 0) {
  XReg mask = X[xs1];
  csr_sw_write(csr_handle, initial_csr_value & ~mask);
}
X[xd] = initial_csr_value;
```

B.12.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

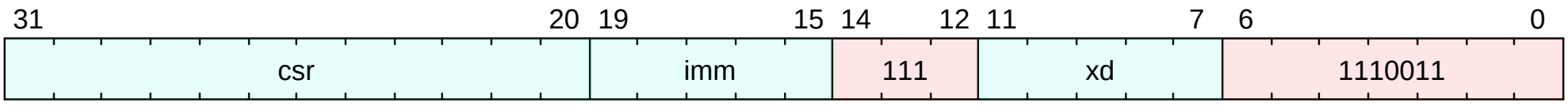
B.13. csrrci

Atomic Read and Clear Bits in CSR with Immediate

This instruction is defined by:

Zicsr

B.13.1. Encoding



B.13.2. Description

The CSRRCI variant is similar to CSRRC, except this updates the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (imm[4:0]) field encoded in the `xs1` field instead of a value from an integer register. For CSRRCI, if the `imm[4:0]` field is zero, then this instruction will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write, nor raise illegal-instruction exceptions on accesses to read-only CSRs. The CSRRCI will always read the CSR and cause any read side effects regardless of `xd` and `xs1` fields.

B.13.3. Access

M
Always

B.13.4. Decode Variables

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

B.13.5. IDL Operation

```
Boolean will_write = imm != 0;
Csr csr_handle = direct_csr_lookup(csr);
if (csr_handle.valid == false) {
  unimplemented_csr($encoding);
} else if (!compatible_mode?(csr_handle.mode, mode())) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if (will_write && csr_handle.writable == false) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg initial_csr_value = csr_sw_read(csr_handle);
if (will_write) {
  XReg mask = imm;
  csr_sw_write(csr_handle, initial_csr_value & ~mask);
}
X[xd] = initial_csr_value;
```

B.13.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

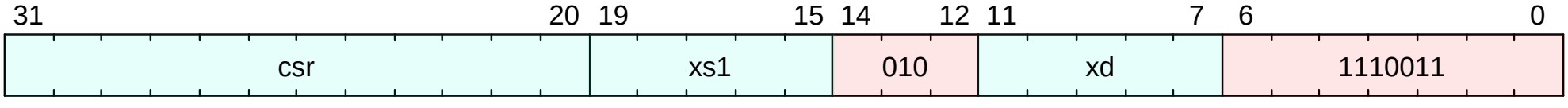
B.14. csrrs

Atomic Read and Set Bits in CSR

This instruction is defined by:

Zicsr

B.14.1. Encoding



B.14.2. Description

Atomically read and set bits in a CSR.

Reads the value of the CSR, zero-extends the value to `XLEN` bits, and writes it to integer register `xd`. The initial value in integer register `xs1` is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in `xs1` will cause the corresponding bit to be set in the CSR, if that CSR bit is writable. Other bits in the CSR are not explicitly written.

B.14.3. Access

M
Always

B.14.4. Decode Variables

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

B.14.5. IDL Operation

```
Boolean will_write = xs1 != 0;
Csr csr_handle = direct_csr_lookup(csr);
if (csr_handle.valid == false) {
  unimplemented_csr($encoding);
} else if (!compatible_mode?(csr_handle.mode, mode())) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if (will_write && csr_handle.writable == false) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg initial_csr_value = csr_sw_read(csr_handle);
if (will_write) {
  XReg mask = X[xs1];
  csr_sw_write(csr_handle, initial_csr_value | mask);
}
X[xd] = initial_csr_value;
```

B.14.6. Sail Operation

```
{
  let rs1_val : xlenbits = if is_imm then zero_extend(rs1) else X(rs1);
  let isWrite : bool = match op {
    CSRRW  => true,
    _      => if is_imm then unsigned(rs1_val) != 0 else unsigned(rs1) != 0
  };
  if not(check_CSR(csr, cur_privilege, isWrite))
  then { handle_illegal(); RETIRE_FAIL }
  else if not(ext_check_CSR(csr, cur_privilege, isWrite))
  then { ext_check_CSR_fail(); RETIRE_FAIL }
  else {
    let csr_val = readCSR(csr); /* could have side-effects, so technically shouldn't perform for CSRW[I] with rd == 0 */
    if isWrite then {
      let new_val : xlenbits = match op {
        CSRRW => rs1_val,
        CSRRS => csr_val | rs1_val,
```

```
        CSRRC => csr_val & ~(rs1_val)
      };
      writeCSR(csr, new_val)
    };
    X(rd) = csr_val;
    RETIRE_SUCCESS
  }
}
```

B.14.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

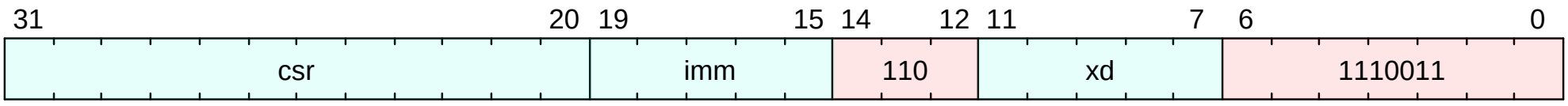
B.15. csrrsi

Atomic Read and Set Bits in CSR with Immediate

This instruction is defined by:

Zicsr

B.15.1. Encoding



B.15.2. Description

The CSRRSI variant is similar to CSRRS, except this updates the CSR using an XLEN-bit value obtained by zero-extending a 5-bit unsigned immediate (imm[4:0]) field encoded in the `xs1` field instead of a value from an integer register. For CSRRSI, if the `imm[4:0]` field is zero, then this instruction will not write to the CSR, and shall not cause any of the side effects that might otherwise occur on a CSR write, nor raise illegal-instruction exceptions on accesses to read-only CSRs. The CSRRSI will always read the CSR and cause any read side effects regardless of `xd` and `xs1` fields.

B.15.3. Access

M
Always

B.15.4. Decode Variables

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

B.15.5. IDL Operation

```
Boolean will_write = imm != 0;
Csr csr_handle = direct_csr_lookup(csr);
if (csr_handle.valid == false) {
  unimplemented_csr($encoding);
} else if (!compatible_mode?(csr_handle.mode, mode())) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if (will_write && csr_handle.writable == false) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg initial_csr_value = csr_sw_read(csr_handle);
if (will_write) {
  XReg mask = imm;
  csr_sw_write(csr_handle, initial_csr_value | mask);
}
X[xd] = initial_csr_value;
```

B.15.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

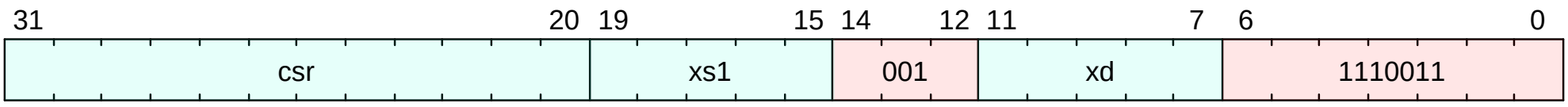
B.16. csrrw

Atomic Read/Write CSR

This instruction is defined by:

Zicsr

B.16.1. Encoding



B.16.2. Description

Atomically swap values in the CSRs and integer registers.

Read the old value of the CSR, zero-extends the value to `XLEN` bits, and then write it to integer register xd. The initial value in xs1 is written to the CSR. If `xd=x0`, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read.

B.16.3. Access

M
Always

B.16.4. Decode Variables

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

B.16.5. IDL Operation

```
Csr csr_handle = direct_csr_lookup(csr);
Bits<MXLEN> initial_value = X[xs1];
if (csr_handle.valid == false) {
  unimplemented_csr($encoding);
} else if (!compatible_mode?(csr_handle.mode, mode())) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if (csr_handle.writable == false) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xd != 0) {
  X[xd] = csr_sw_read(csr_handle);
}
csr_sw_write(csr_handle, initial_value);
```

B.16.6. Sail Operation

```
{
  let rs1_val : xlenbits = if is_imm then zero_extend(rs1) else X(rs1);
  let isWrite : bool = match op {
    CSRRW  => true,
    _      => if is_imm then unsigned(rs1_val) != 0 else unsigned(rs1) != 0
  };
  if not(check_CSR(csr, cur_privilege, isWrite))
  then { handle_illegal(); RETIRE_FAIL }
  else if not(ext_check_CSR(csr, cur_privilege, isWrite))
  then { ext_check_CSR_fail(); RETIRE_FAIL }
  else {
    let csr_val = readCSR(csr); /* could have side-effects, so technically shouldn't perform for CSRW[I] with rd == 0 */
    if isWrite then {
      let new_val : xlenbits = match op {
        CSRRW => rs1_val,
        CSRRS => csr_val | rs1_val,
        CSRRC => csr_val & ~(rs1_val)
      };
      writeCSR(csr, new_val)
```

```
    };
    X(rd) = csr_val;
    RETIRE_SUCCESS
  }
}
```

B.16.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

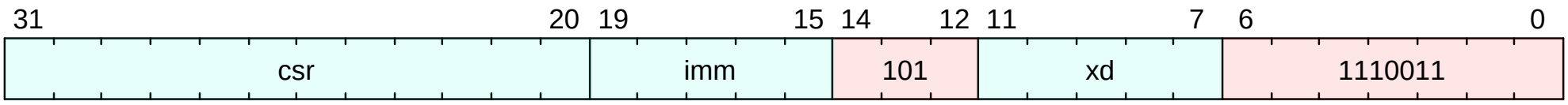
B.17. csrrwi

Atomic Read/Write CSR Immediate

This instruction is defined by:

Zicsr

B.17.1. Encoding



B.17.2. Description

Atomically write CSR using a 5-bit immediate, and load the previous value into 'xd'.

Read the old value of the CSR, zero-extends the value to `XLEN` bits, and then write it to integer register xd. The 5-bit uimm field is zero-extended and written to the CSR. If `xd=x0`, then the instruction shall not read the CSR and shall not cause any of the side effects that might occur on a CSR read.

B.17.3. Access

M
Always

B.17.4. Decode Variables

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

B.17.5. IDL Operation

```
Csr csr_handle = direct_csr_lookup(csr);
if (csr_handle.valid == false) {
  unimplemented_csr($encoding);
} else if (!compatible_mode?(csr_handle.mode, mode())) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if (csr_handle.writable == false) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xd != 0) {
  X[xd] = csr_sw_read(csr_handle);
}
csr_sw_write(csr_handle, {{MXLEN - 5{1'b0}}, imm});
```

B.17.6. Sail Operation

```
{
  let rs1_val : xlenbits = if is_imm then zero_extend(rs1) else X(rs1);
  let isWrite : bool = match op {
    CSRRW  => true,
    _      => if is_imm then unsigned(rs1_val) != 0 else unsigned(rs1) != 0
  };
  if not(check_CSR(csr, cur_privilege, isWrite))
  then { handle_illegal(); RETIRE_FAIL }
  else if not(ext_check_CSR(csr, cur_privilege, isWrite))
  then { ext_check_CSR_fail(); RETIRE_FAIL }
  else {
    let csr_val = readCSR(csr); /* could have side-effects, so technically shouldn't perform for CSRW[I] with rd == 0 */
    if isWrite then {
      let new_val : xlenbits = match op {
        CSRRW => rs1_val,
        CSRRS => csr_val | rs1_val,
        CSRRC => csr_val & ~(rs1_val)
      };
      writeCSR(csr, new_val)
    };
```

```
    X(rd) = csr_val;
    RETIRE_SUCCESS
  }
}
```

B.17.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

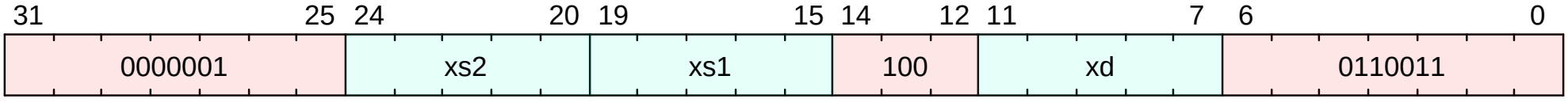
B.18. div

Signed division

This instruction is defined by:

M

B.18.1. Encoding



B.18.2. 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.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::M) && (CSR[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.18.6. 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.18.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

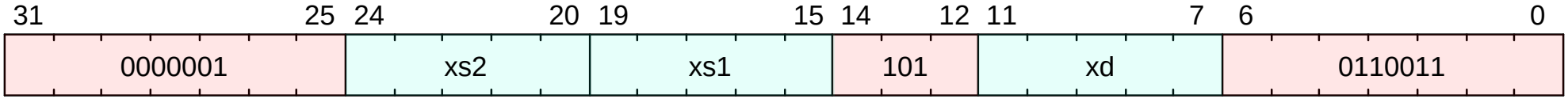
B.19. divu

Unsigned division

This instruction is defined by:

M

B.19.1. Encoding



B.19.2. 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.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::M) && (CSR[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.19.6. 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.19.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

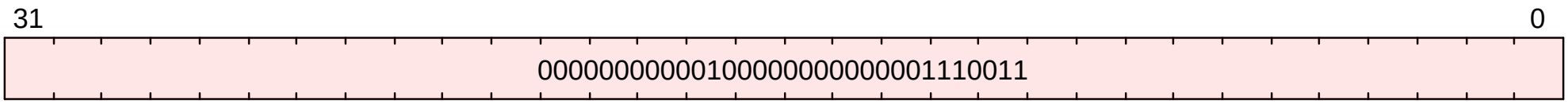
B.20. ebreak

Breakpoint exception

This instruction is defined by:

I

B.20.1. Encoding



B.20.2. 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.20.3. Access

M
Always

B.20.4. Decode Variables

```
```

B.20.5. IDL Operation

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

B.20.6. Sail Operation

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

B.20.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

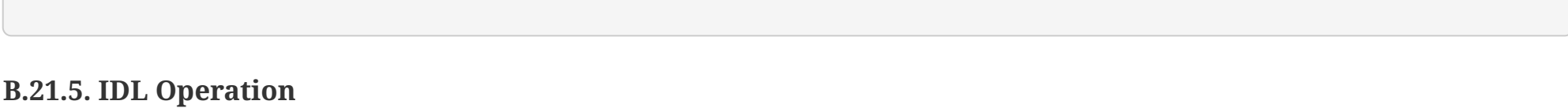
B.21. ecall

Environment call

This instruction is defined by:

I

B.21.1. Encoding



B.21.2. 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.21.3. Access

M
Always

B.21.4. Decode Variables

```
```

B.21.5. 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.21.6. 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.21.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

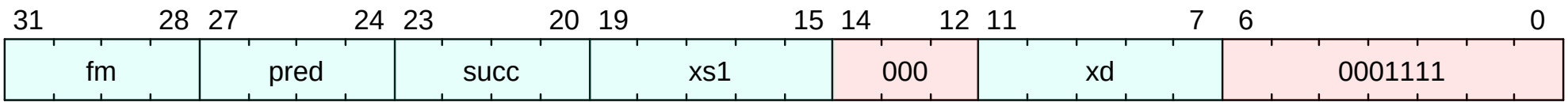
B.22. fence

Memory ordering fence

This instruction is defined by:

I

B.22.1. Encoding



B.22.2. 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 6. 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 7. 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 8. 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.22.3. Access

M
Always

B.22.4. 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.22.5. IDL Operation

```
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 (mode() == PrivilegeMode::S) {
  if (CSR[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 ((CSR[menvcfg].FIOM | CSR[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 ((CSR[menvcfg].FIOM | CSR[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.22.6. 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.22.7. Exceptions

This instruction does not generate synchronous exceptions.

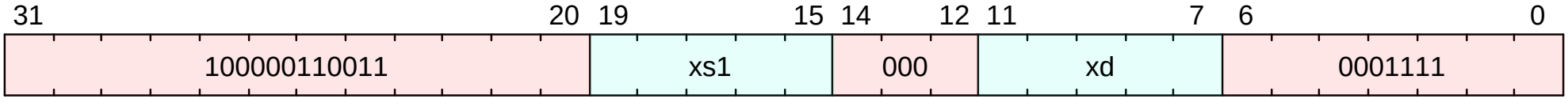
B.23. fence.tso

Memory ordering fence, total store ordering

This instruction is defined by:

I

B.23.1. Encoding



B.23.2. 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.23.3. Access

M
Always

B.23.4. Decode Variables

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

B.23.5. IDL Operation

```
fence_tso();
```

B.23.6. 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.23.7. Exceptions

This instruction does not generate synchronous exceptions.

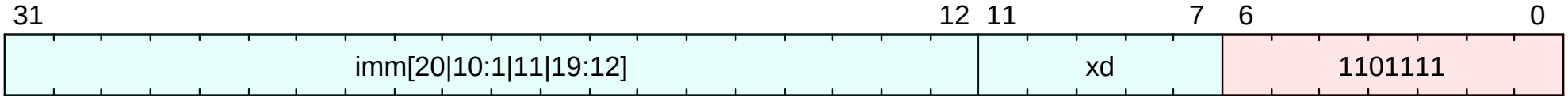
B.24. jal

Jump and link

This instruction is defined by:

I

B.24.1. Encoding



B.24.2. Description

Jump to a PC-relative offset and store the return address in xd.

B.24.3. Access

M
Always

B.24.4. 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.24.5. IDL Operation

```
XReg return_addr = $pc + 4;
X[xd] = return_addr;
jump_halfword($pc + $signed(imm));
```

B.24.6. 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.24.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

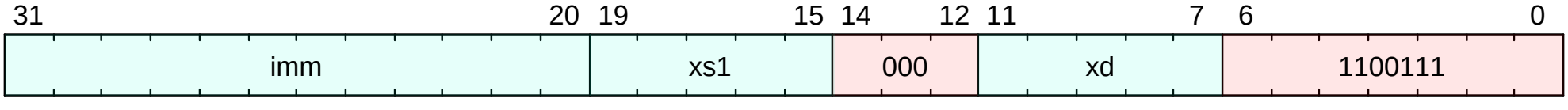
B.25. jalr

Jump and link register

This instruction is defined by:

I

B.25.1. Encoding



B.25.2. 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.25.3. Access

M
Always

B.25.4. Decode Variables

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

B.25.5. IDL Operation

```
XReg addr = (X[xs1] + $signed(imm)) & ~MXLEN'1;
XReg returnaddr;
returnaddr = $pc + 4;
X[xd] = returnaddr;
jump(addr);
```

B.25.6. 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.25.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

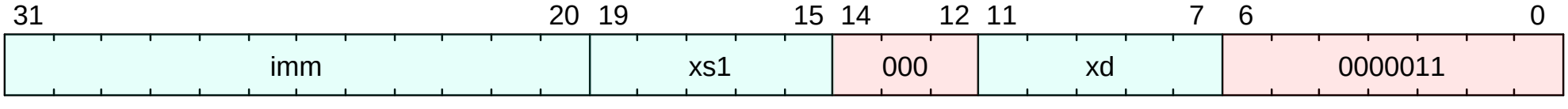
B.26. lb

Load byte

This instruction is defined by:

I

B.26.1. Encoding



B.26.2. 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.26.3. Access

M
Always

B.26.4. Decode Variables

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

B.26.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = sext(read_memory<8>(virtual_address, $encoding), 8);
```

B.26.6. 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.26.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault
- StoreAmoAccessFault

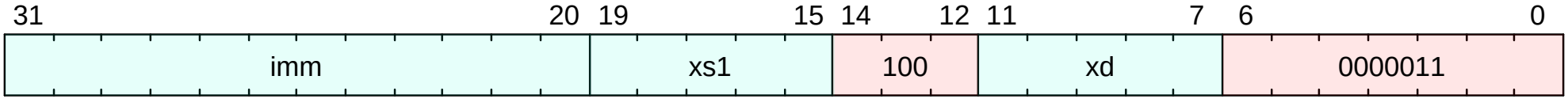
B.27. lbu

Load byte unsigned

This instruction is defined by:

I

B.27.1. Encoding



B.27.2. 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.27.3. Access

M
Always

B.27.4. Decode Variables

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

B.27.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = read_memory<8>(virtual_address, $encoding);
```

B.27.6. 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.27.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

B.28. ld

Load doubleword

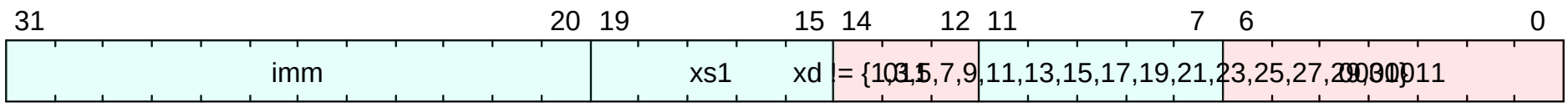
This instruction is defined by:

(I || Zilsd)

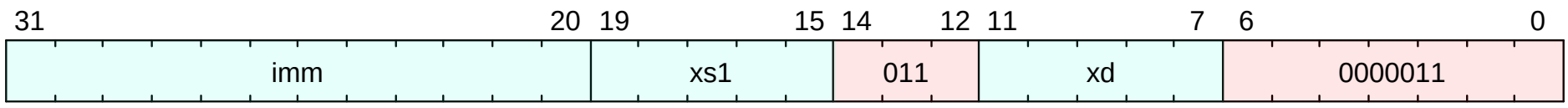
B.28.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.28.2. 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.28.3. Access

M
Always

B.28.4. 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.28.5. 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.28.6. 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.28.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

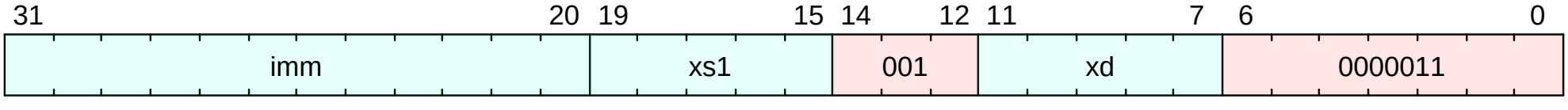
B.29. lh

Load halfword

This instruction is defined by:

I

B.29.1. Encoding



B.29.2. 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.29.3. Access

M
Always

B.29.4. Decode Variables

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

B.29.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = sext(read_memory<16>(virtual_address, $encoding), 16);
```

B.29.6. 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.29.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

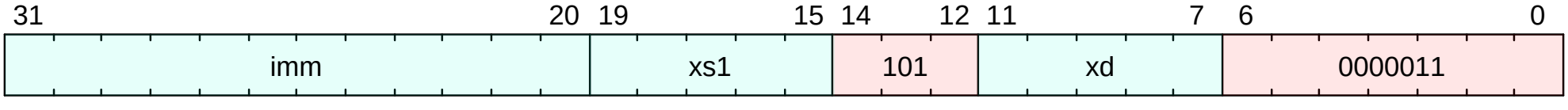
B.30. lhu

Load halfword unsigned

This instruction is defined by:

I

B.30.1. Encoding



B.30.2. 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.30.3. Access

M
Always

B.30.4. Decode Variables

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

B.30.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = read_memory<16>(virtual_address, $encoding);
```

B.30.6. 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.30.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

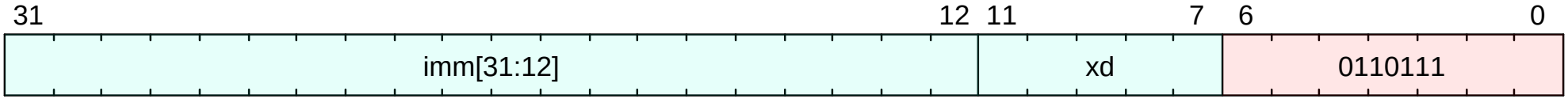
B.31. lui

Load upper immediate

This instruction is defined by:

I

B.31.1. Encoding



B.31.2. Description

Load the zero-extended imm into xd.

B.31.3. Access

M
Always

B.31.4. Decode Variables

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

B.31.5. IDL Operation

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

B.31.6. 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.31.7. Exceptions

This instruction does not generate synchronous exceptions.

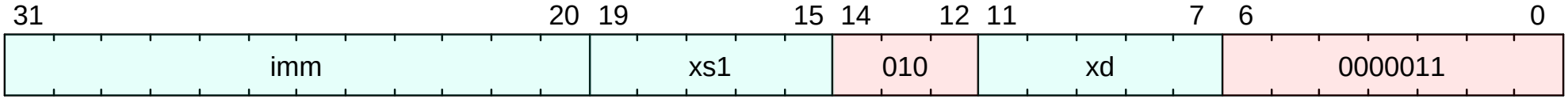
B.32. lw

Load word

This instruction is defined by:

I

B.32.1. Encoding



B.32.2. 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.32.3. Access

M
Always

B.32.4. Decode Variables

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

B.32.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = sext(read_memory<32>(virtual_address, $encoding), 32);
```

B.32.6. 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.32.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

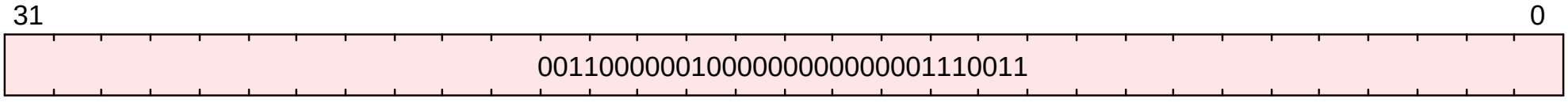
B.33. mret

Machine-mode Return from Trap

This instruction is defined by:

Sm

B.33.1. Encoding



B.33.2. Description

Return from machine mode after handling a trap.

B.33.3. Access

M
Always

B.33.4. Decode Variables

```
```

B.33.5. IDL Operation

```
if (CSR[mstatus].MPP != 2'b11) {
  if (implemented?(ExtensionName::U)) {
    CSR[mstatus].MPRV = 0;
  }
}
if (implemented?(ExtensionName::Smdbltrp)) {
  if (xlen() == 64) {
    CSR[mstatus].MDT = 1'b0;
  } else {
    CSR[mstatush].MDT = 1'b0;
  }
}
CSR[mstatus].MIE = CSR[mstatus].MPIE;
CSR[mstatus].MPIE = 1;
if (CSR[mstatus].MPP == 2'b00) {
  set_mode(PrivilegeMode::U);
} else if (CSR[mstatus].MPP == 2'b01) {
  set_mode(PrivilegeMode::S);
} else if (CSR[mstatus].MPP == 2'b11) {
  set_mode(PrivilegeMode::M);
}
CSR[mstatus].MPP = implemented?(ExtensionName::U) ? 2'b00 : 2'b11;
$pc = $bits(CSR[CSR[mepc]]);
```

B.33.6. Sail Operation

```
{
  if   cur_privilege != Machine
  then { handle_illegal(); RETIRE_FAIL }
  else if not(ext_check_xret_priv (Machine))
  then { ext_fail_xret_priv(); RETIRE_FAIL }
  else {
    set_next_pc(exception_handler(cur_privilege, CTL_MRET(), PC));
    RETIRE_SUCCESS
  }
}
```

B.33.7. Exceptions

This instruction does not generate synchronous exceptions.

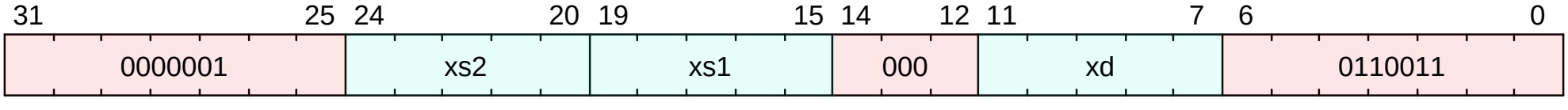
B.34. mul

Signed multiply

This instruction is defined by:

(M || Zmmul)

B.34.1. Encoding



B.34.2. 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.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::M) && (CSR[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.34.6. 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.34.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

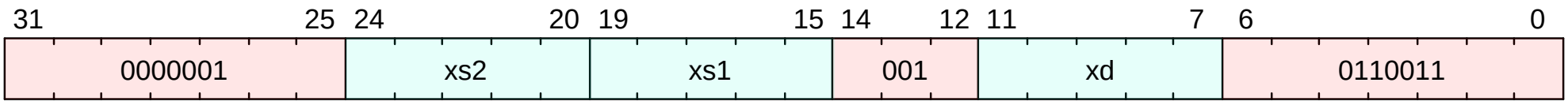
B.35. mulh

Signed multiply high

This instruction is defined by:

(M || Zmmul)

B.35.1. Encoding



B.35.2. 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.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::M) && (CSR[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.35.6. 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.35.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

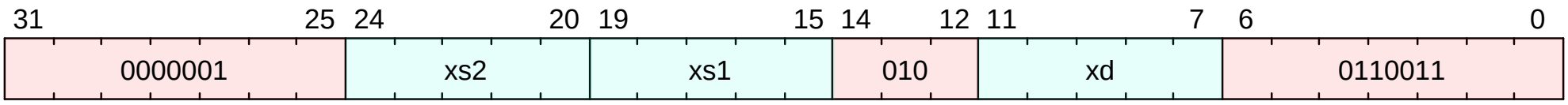
B.36. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

(M || Zmmul)

B.36.1. Encoding



B.36.2. 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.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::M) && (CSR[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.36.6. 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.36.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

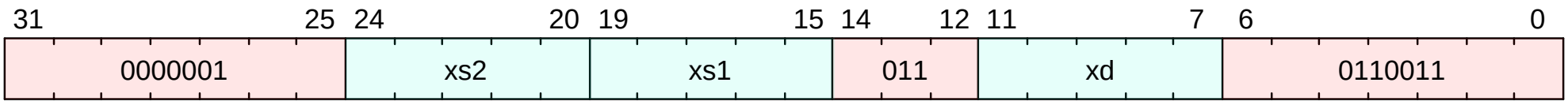
B.37. mulhu

Unsigned multiply high

This instruction is defined by:

(M || Zmmul)

B.37.1. Encoding



B.37.2. 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.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::M) && (CSR[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.37.6. 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.37.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

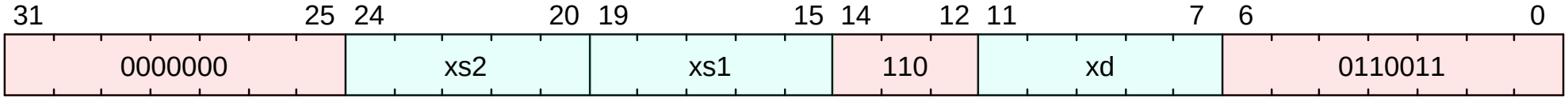
B.38. or

Or

This instruction is defined by:

I

B.38.1. Encoding



B.38.2. Description

Or xs1 with xs2, and store the result in xd

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

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

B.38.6. 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.38.7. Exceptions

This instruction does not generate synchronous exceptions.

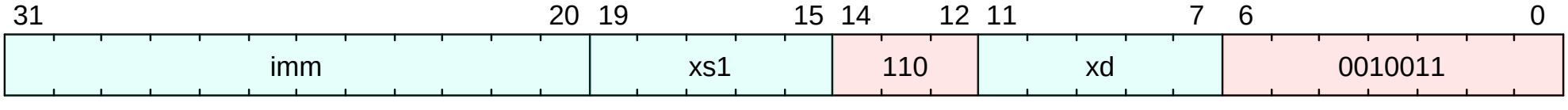
B.39. ori

Or immediate

This instruction is defined by:

I

B.39.1. Encoding



B.39.2. Description

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

B.39.3. Access

M
Always

B.39.4. Decode Variables

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

B.39.5. IDL Operation

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

B.39.6. 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.39.7. Exceptions

This instruction does not generate synchronous exceptions.

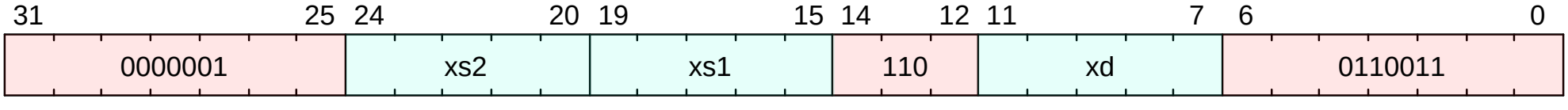
B.40. rem

Signed remainder

This instruction is defined by:

M

B.40.1. Encoding



B.40.2. 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.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::M) && (CSR[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.40.6. 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.40.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

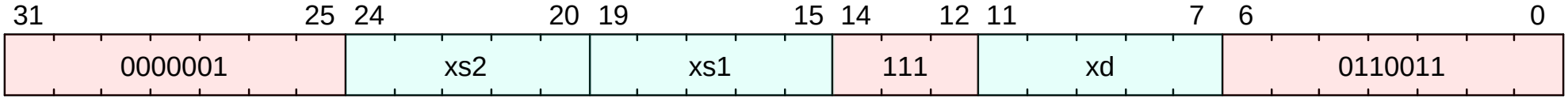
B.41. remu

Unsigned remainder

This instruction is defined by:

M

B.41.1. Encoding



B.41.2. Description

Calculate the remainder of unsigned division of xs1 by xs2, and store the result in xd.

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::M) && (CSR[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.41.6. 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.41.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

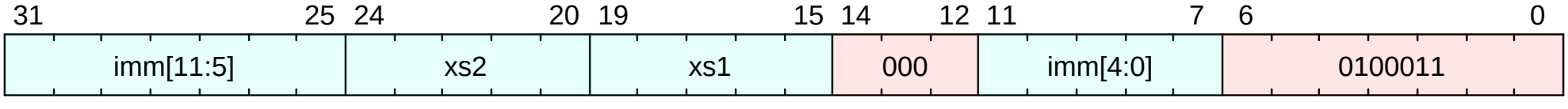
B.42. sb

Store byte

This instruction is defined by:

I

B.42.1. Encoding



B.42.2. Description

Store 8 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.42.3. Access

M
Always

B.42.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.42.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<8>(virtual_address, X[xs2][7:0], $encoding);
```

B.42.6. 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.42.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

B.43. sd

Store doubleword

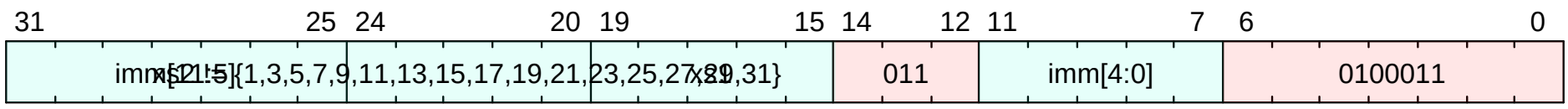
This instruction is defined by:

(I || Zilsd)

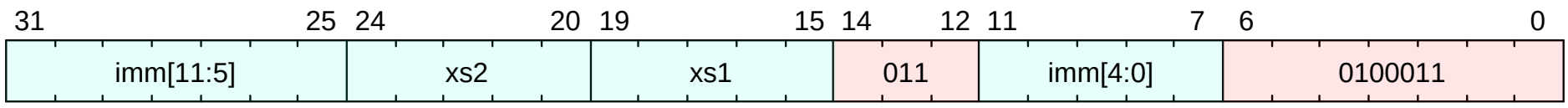
B.43.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.43.2. 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.43.3. Access

M
Always

B.43.4. 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.43.5. 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.43.6. 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.43.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

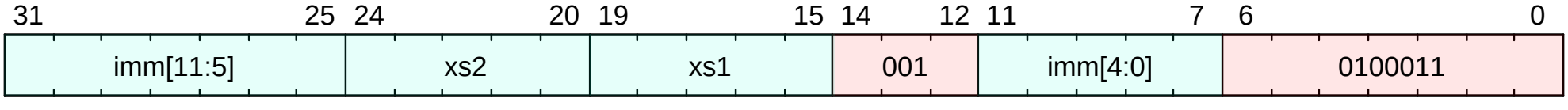
B.44. sh

Store halfword

This instruction is defined by:

I

B.44.1. Encoding



B.44.2. Description

Store 16 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.44.3. Access

M
Always

B.44.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.44.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<16>(virtual_address, X[xs2][15:0], $encoding);
```

B.44.6. 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.44.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

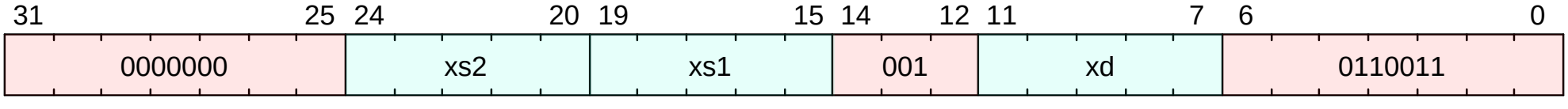
B.45. sll

Shift left logical

This instruction is defined by:

I

B.45.1. Encoding



B.45.2. Description

Shift the value in `xs1` left by the value in the lower 6 bits of `xs2`, and store the result in `xd`.

B.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 (xlen() == 64) {
  X[xd] = X[xs1] << X[xs2][5:0];
} else {
  X[xd] = X[xs1] << X[xs2][4:0];
}
```

B.45.6. 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.45.7. Exceptions

This instruction does not generate synchronous exceptions.

B.46. slli

Shift left logical immediate

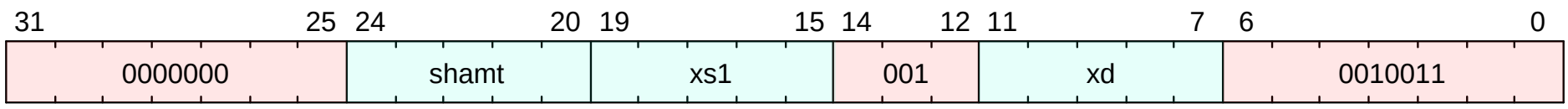
This instruction is defined by:

I

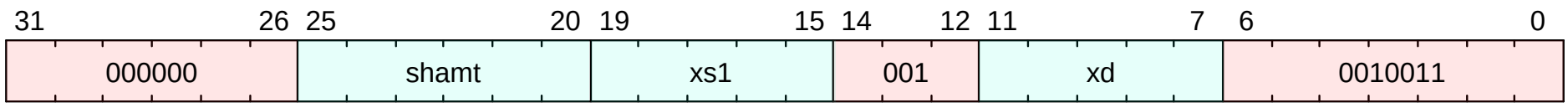
B.46.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.46.2. Description

Shift the value in xs1 left by shamt, and store the result in xd

B.46.3. Access

M
Always

B.46.4. 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.46.5. IDL Operation

```
X[xd] = X[xs1] << shamt;
```

B.46.6. 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.46.7. Exceptions

This instruction does not generate synchronous exceptions.

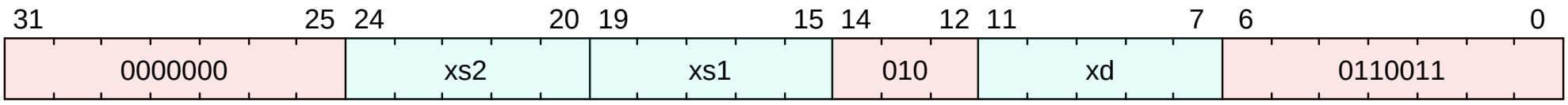
B.47. slt

Set on less than

This instruction is defined by:

I

B.47.1. Encoding



B.47.2. 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.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

```
XReg src1 = X[xs1];
XReg src2 = X[xs2];
X[xd] = ($signed(src1) < $signed(src2)) ? '1 : '0;
```

B.47.6. 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.47.7. Exceptions

This instruction does not generate synchronous exceptions.

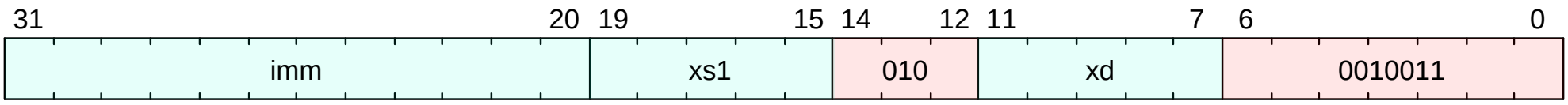
B.48. slti

Set on less than immediate

This instruction is defined by:

I

B.48.1. Encoding



B.48.2. 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.48.3. Access

M
Always

B.48.4. Decode Variables

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

B.48.5. IDL Operation

```
X[xd] = ($signed(X[xs1]) < $signed(imm)) ? '1 : '0;
```

B.48.6. 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.48.7. Exceptions

This instruction does not generate synchronous exceptions.

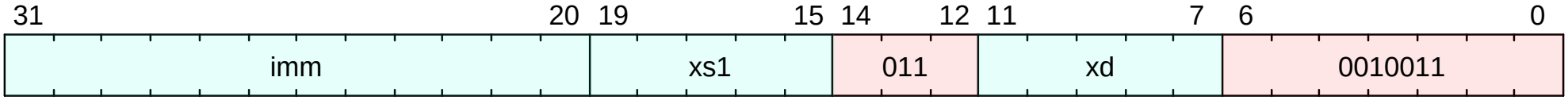
B.49. sltiu

Set on less than immediate unsigned

This instruction is defined by:

I

B.49.1. Encoding



B.49.2. 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.49.3. Access

M
Always

B.49.4. Decode Variables

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

B.49.5. IDL Operation

```
Bits<MXLEN> sign_extend_imm = $signed(imm);
X[xd] = (X[xs1] < sign_extend_imm) ? 1 : 0;
```

B.49.6. 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.49.7. Exceptions

This instruction does not generate synchronous exceptions.

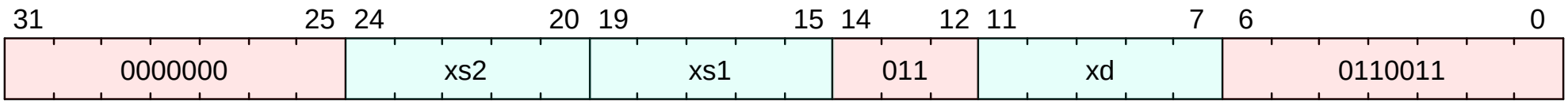
B.50. sltu

Set on less than unsigned

This instruction is defined by:

I

B.50.1. Encoding



B.50.2. 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.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

```
X[xd] = (X[xs1] < X[xs2]) ? 1 : 0;
```

B.50.6. 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.50.7. Exceptions

This instruction does not generate synchronous exceptions.

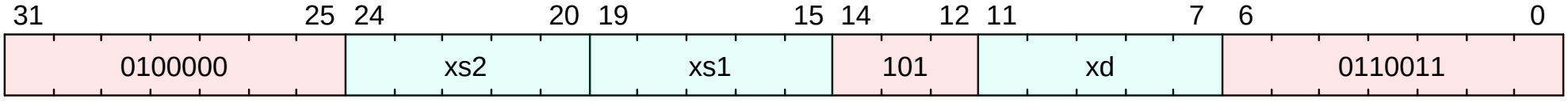
B.51. sra

Shift right arithmetic

This instruction is defined by:

I

B.51.1. Encoding



B.51.2. 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.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 (xlen() == 64) {
  X[xd] = X[xs1] >>> X[xs2][5:0];
} else {
  X[xd] = X[xs1] >>> X[xs2][4:0];
}
```

B.51.6. 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.51.7. Exceptions

This instruction does not generate synchronous exceptions.

B.52. srai

Shift right arithmetic immediate

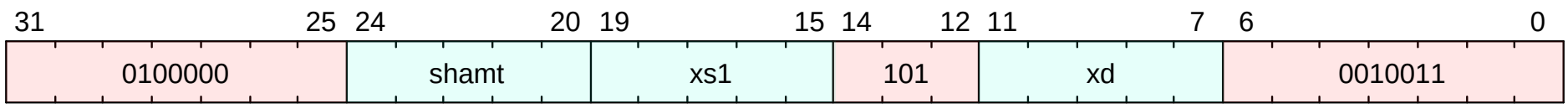
This instruction is defined by:

I

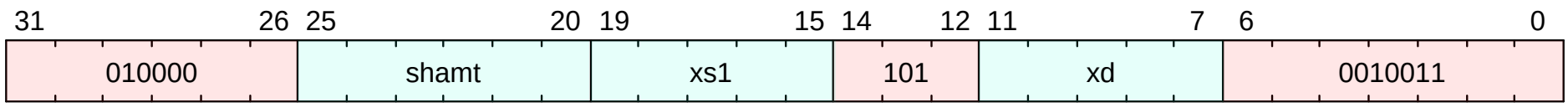
B.52.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.52.2. 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.52.3. Access

M
Always

B.52.4. 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.52.5. IDL Operation

```
X[xd] = X[xs1] >>> shamt;
```

B.52.6. 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.52.7. Exceptions

This instruction does not generate synchronous exceptions.

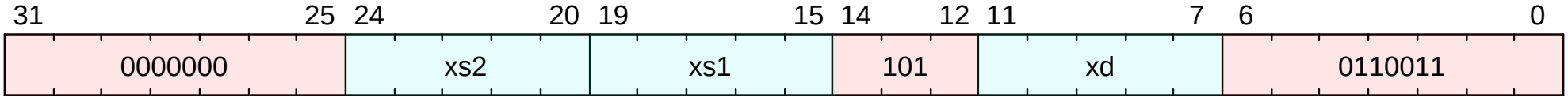
B.53. srl

Shift right logical

This instruction is defined by:

I

B.53.1. Encoding



B.53.2. Description

Logical shift the value in `xs1` right by the value in the lower bits of `xs2`, and store the result in `xd`.

B.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 (xlen() == 64) {
  X[xd] = X[xs1] >> X[xs2][5:0];
} else {
  X[xd] = X[xs1] >> X[xs2][4:0];
}
```

B.53.6. 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.53.7. Exceptions

This instruction does not generate synchronous exceptions.

B.54. srli

Shift right logical immediate

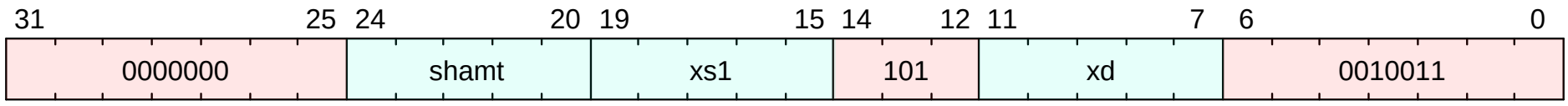
This instruction is defined by:

I

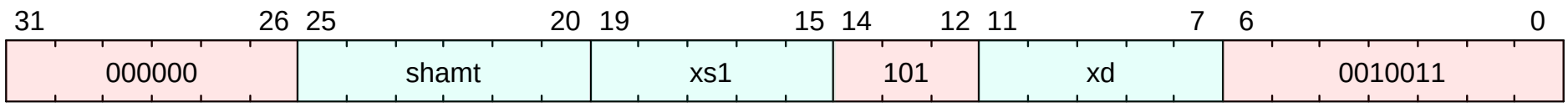
B.54.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.54.2. Description

Shift the value in xs1 right by shamt, and store the result in xd

B.54.3. Access

M
Always

B.54.4. 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.54.5. IDL Operation

```
X[xd] = X[xs1] >> shamt;
```

B.54.6. 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.54.7. Exceptions

This instruction does not generate synchronous exceptions.

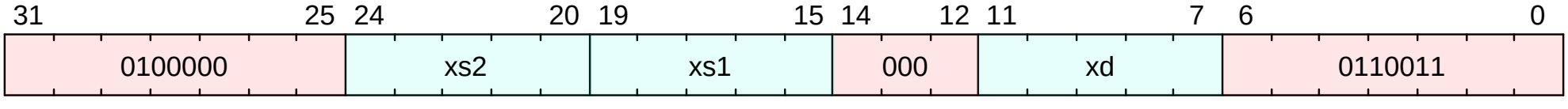
B.55. sub

Subtract

This instruction is defined by:

I

B.55.1. Encoding



B.55.2. Description

Subtract the value in xs2 from xs1, and store the result in xd

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

```
XReg t0 = X[xs1];
XReg t1 = X[xs2];
X[xd] = t0 - t1;
```

B.55.6. 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.55.7. Exceptions

This instruction does not generate synchronous exceptions.

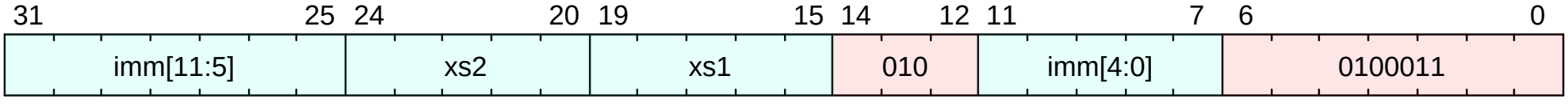
B.56. sw

Store word

This instruction is defined by:

I

B.56.1. Encoding



B.56.2. Description

Store 32 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.56.3. Access

M
Always

B.56.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> xs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

B.56.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<32>(virtual_address, X[xs2][31:0], $encoding);
```

B.56.6. 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.56.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

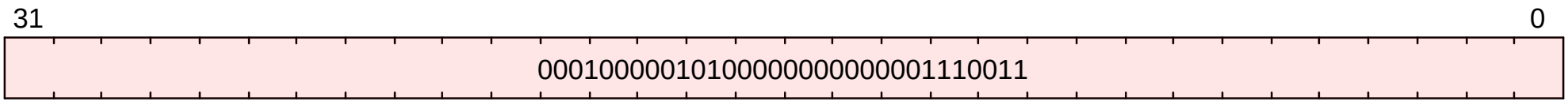
B.57. wfi

Wait for interrupt

This instruction is defined by:

Sm

B.57.1. Encoding



B.57.2. Description

Can causes the processor to enter a low-power state until the next interrupt occurs.

<%- if ext?(:H) -%> The behavior of wfi is affected by the mstatus.TW and hstatus.VTW bits, as summarized below.

mstatus.TW	hstatus.VTW	wfi behavior			
		HS-mode	**U-mode**	**VS-mode**	**in VU-mode**
0	0	Wait	Trap (I)	Wait	Trap (V)
0	1	Wait	Trap (I)	Trap (V)	Trap (V)
1	-	Trap (I)	Trap (I)	Trap (I)	Trap (I)
Trap (I) - Trap with `Illegal Instruction` code Trap (V) - Trap with `Virtual Instruction` code					

<%- else -%> The wfi instruction is also affected by mstatus.TW, as shown below:

mstatus.TW	wfi behavior	
	S-mode	**U-mode**
0	Wait	Trap (I)
1	Trap (I)	Trap (I)
Trap (I) - Trap with `Illegal Instruction` code		

<%- end -%>

When wfi is marked as causing a trap above, the implementation is allowed to wait for an unspecified period of time to see if an interrupt occurs before raising the trap. That period of time can be zero (*i.e.*, wfi always causes a trap in the cases identified above).

B.57.3. Access

M
Always

<%- if ext?(:H) -%> The behavior of wfi is affected by the mstatus.TW and hstatus.VTW bits, as summarized below.

mstatus.TW	hstatus.VTW	wfi behavior			
		HS-mode	**U-mode**	**VS-mode**	**in VU-mode**
0	0	Wait	Trap (I)	Wait	Trap (V)
0	1	Wait	Trap (I)	Trap (V)	Trap (V)
1	-	Trap (I)	Trap (I)	Trap (I)	Trap (I)
Trap (I) - Trap with `Illegal Instruction` code Trap (V) - Trap with `Virtual Instruction` code					

<%- else -%> The wfi instruction is also affected by mstatus.TW, as shown below:

mstatus.TW	wfi behavior	
	S-mode	**U-mode**
0	Wait	Trap (I)
1	Trap (I)	Trap (I)
Trap (I) - Trap with `Illegal Instruction` code		

<%- end -%>

B.57.4. Decode Variables

```
```

B.57.5. IDL Operation

```
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if ((CSR[misa].S == 1) && (CSR[mstatus].TW == 1'b1)) {
  if (mode() != PrivilegeMode::M) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
if (CSR[misa].H == 1) {
  if (CSR[hstatus].VTW == 1'b0) {
    if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (CSR[hstatus].VTW == 1'b1) {
    if ((mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU)) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  }
}
wfi();
```

B.57.6. Sail Operation

```
match cur_privilege {
    Machine    => { platform_wfi(); RETIRE_SUCCESS },
    Supervisor => if   mstatus.TW() == 0b1
                  then { handle_illegal(); RETIRE_FAIL }
                  else { platform_wfi(); RETIRE_SUCCESS },
    User       => { handle_illegal(); RETIRE_FAIL }
  }
```

B.57.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

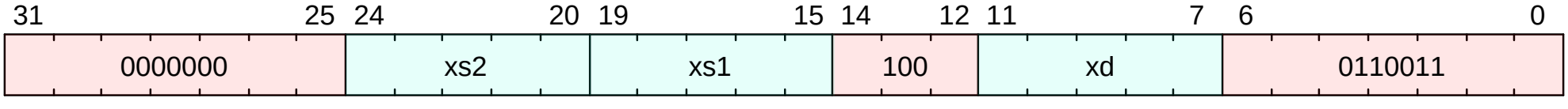
B.58. xor

Exclusive Or

This instruction is defined by:

I

B.58.1. Encoding



B.58.2. Description

Exclusive or xs1 with xs2, and store the result in xd

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

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

B.58.6. 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.58.7. Exceptions

This instruction does not generate synchronous exceptions.

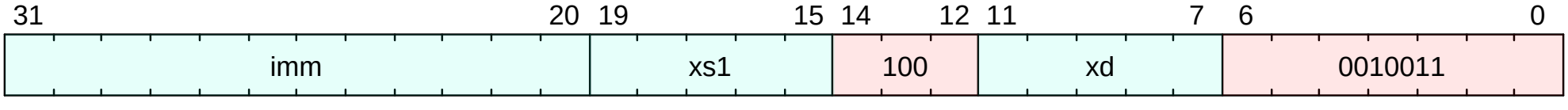
B.59. xori

Exclusive Or immediate

This instruction is defined by:

I

B.59.1. Encoding



B.59.2. Description

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

B.59.3. Access

M
Always

B.59.4. Decode Variables

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

B.59.5. IDL Operation

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

B.59.6. 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.59.7. 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
Length	64-bit
Privilege Mode	U

C.1.2. Format

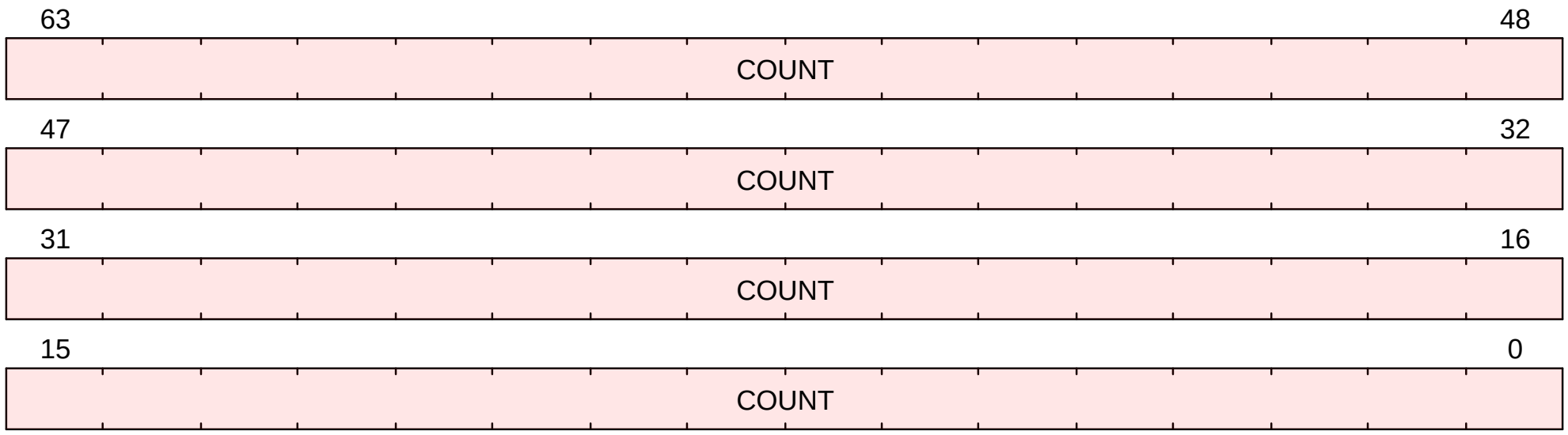


Figure 1. cycle format

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 (CSR[mcounteren].CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].CY & CSR[scounteren].CY) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].CY == 1'b0 && CSR[mcounteren].CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].CY & CSR[scounteren].CY) == 1'b0) && (CSR[mcounteren].CY == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle();
```

C.2. 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.2.1. Attributes

CSR Address	0xc02
Defining extension	Zicntr
Length	64-bit
Privilege Mode	U

C.2.2. Format

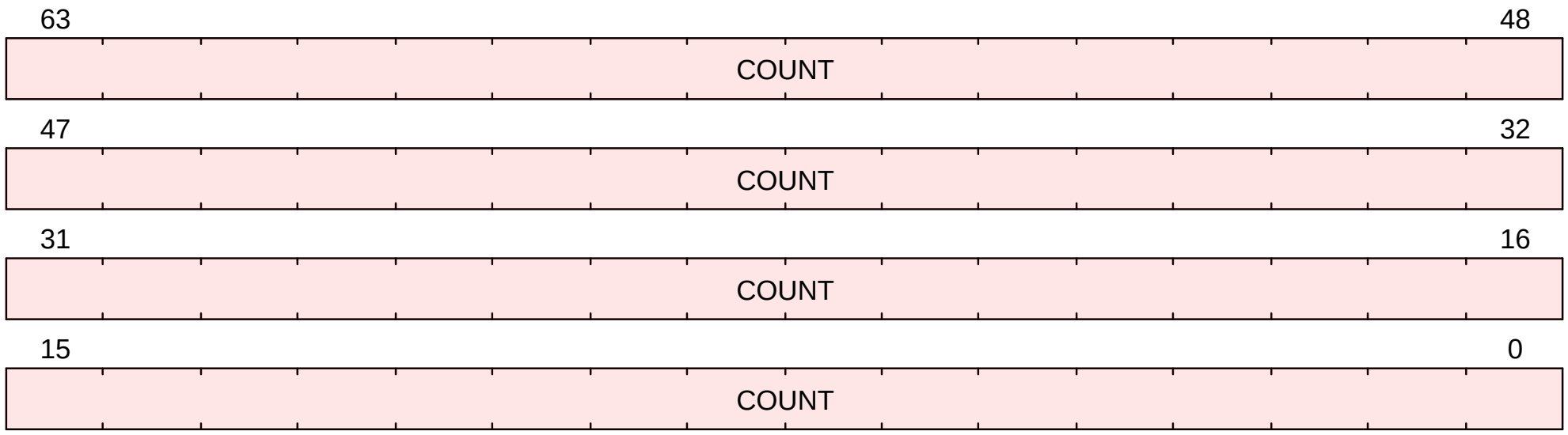


Figure 2. instret format

C.2.3. Field Summary

Name	Location	Type	Reset Value
instret.COUNT	63:0	RO-H	0

C.2.4. Fields

instret.COUNT Field

Location:

63:0

Description:

Alias of minstret.COUNT.

Type:

RO-H

Reset value:

0

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 (CSR[mcounteren].IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].IR & CSR[scounteren].IR) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].IR == 1'b0 && CSR[mcounteren].IR == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].IR & CSR[scounteren].IR) == 1'b0) && (CSR[mcounteren].IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return CSR[minstret].COUNT;
```

C.3. marchid

Machine Architecture ID

The marchid CSR is an MXLEN-bit read-only register encoding the base microarchitecture of the hart. This register must be readable in any implementation, but a value of 0 can be returned to indicate the field is not implemented. The combination of mvendorid and marchid should uniquely identify the type of hart microarchitecture that is implemented.

Open-source project architecture IDs are allocated globally by RISC-V International, and have non-zero architecture IDs with a zero most-significant-bit (MSB). Commercial architecture IDs are allocated by each commercial vendor independently, but must have the MSB set and cannot contain zero in the remaining MXLEN-1 bits.

The intent is for the architecture ID to represent the microarchitecture associated with the repo around which development occurs rather than a particular organization. Commercial fabrications of open-source designs should (and might be required by the license to) retain the original architecture ID. This will aid in reducing fragmentation and tool support costs, as well as provide attribution. Open-source architecture IDs are administered by RISC-V International and should only be allocated to released, functioning open-source projects. Commercial architecture IDs can be managed independently by any registered vendor but are required to have IDs disjoint from the open-source architecture IDs (MSB set) to prevent collisions if a vendor wishes to use both closed-source and open-source microarchitectures.

The convention adopted within the following Implementation field can be used to segregate branches of the same architecture design, including by organization. The misa register also helps distinguish different variants of a design.

C.3.1. Attributes

CSR Address	0xf12
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.3.2. Format

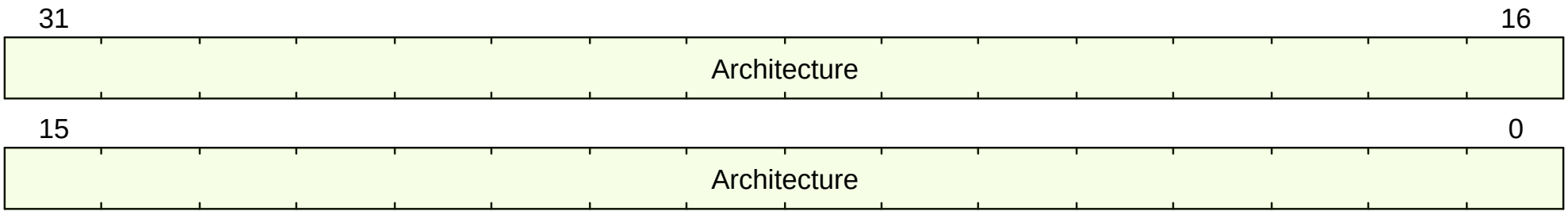


Figure 3. marchid format

C.3.3. Field Summary

Name	Location	Type	Reset Value
marchid.Architecture	31:0	RO	UNDEFINED_LEGAL

C.3.4. Fields

marchid.Architecture Field

Location:

31:0

Description:

Vendor-specific microarchitecture ID.

Type:

RO

Reset value:

UNDEFINED_LEGAL

C.4. mcause

Machine Cause

Reports the cause of the latest exception.

C.4.1. Attributes

CSR Address	0x342
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.4.2. Format

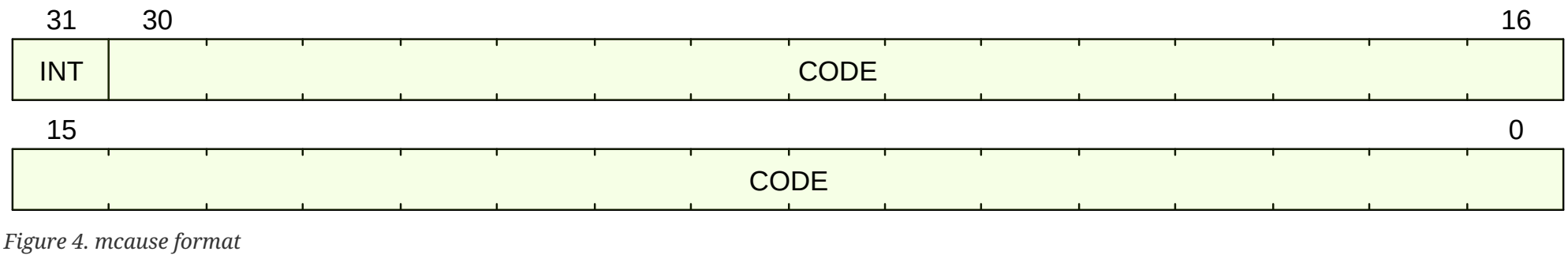


Figure 4. mcause format

C.4.3. Field Summary

Name	Location	Type	Reset Value
mcause.INT	31	RW-RH	0
mcause.CODE	30:0	RW-RH	0

C.4.4. Fields

mcause.INT Field

Location:

31

Description:

Written by hardware when a trap is taken into M-mode.

When set, the last exception was caused by an asynchronous Interrupt.

mcause.INT is writable.

[when,"TRAP_ON_ILLEGAL_WLRL == true"]
If mcause is written with an undefined cause (combination of mcause.INT and mcause.CODE), an `Illegal Instruction` exception occurs.

[when,"TRAP_ON_ILLEGAL_WLRL == false"]
If mcause is written with an undefined cause (combination of mcause.INT and mcause.CODE), neither mcause.INT nor mcause.CODE are modified.

Type:

RW-RH

Reset value:

0

mcause.CODE Field

Location:

30:0

Description:

Written by hardware when a trap is taken into M-mode.

Holds the interrupt or exception code for the last taken trap.

mcause.CODE is writable.

[when,"TRAP_ON_ILLEGAL_WLRL == true"]
If mcause is written with an undefined cause (combination of mcause.INT and mcause.CODE), an `Illegal Instruction` exception occurs.

[when,"TRAP_ON_ILLEGAL_WLRL == false"]
If mcause is written with an undefined cause (combination of mcause.INT and mcause.CODE), neither mcause.INT nor mcause.CODE are modified.

Valid interrupt codes are:
[separator="!"]
!===
<%- implemented_interrupt_codes.sort_by{ |code| code.num }.each do |code| -%>
! <%= code.num %> ! <%= code.name %>
<%- end -%>
!===

Valid exception codes are:
[separator="!"]
!===
<%- implemented_exception_codes.sort_by{ |code| code.num }.each do |code| -%>
! <%= code.num %> ! <%= code.name %>
<%- end -%>
!===

Type:

RW-RH

Reset value:

0

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:

```
INT = # the write only holds if the INT/CODE combination is valid
if (csr_value.INT == 1) {
  if (valid_interrupt_code?(csr_value.CODE)) {
    return 1;
  }
  return ILLEGAL_WLRL;
} else {
  if (valid_exception_code?(csr_value.CODE)) {
    return 1;
  }
  return ILLEGAL_WLRL;
}

CODE = # the write only holds if the INT/CODE combination is valid
if (csr_value.INT == 1) {
  if (valid_interrupt_code?(csr_value.CODE)) {
    return csr_value.CODE;
  }
  return ILLEGAL_WLRL;
} else {
  if (valid_exception_code?(csr_value.CODE)) {
    return csr_value.CODE;
  }
  return ILLEGAL_WLRL;
```

```
}
```

C.5. mcountinhibit

Machine Counter Inhibit

Bits to inhibit (stops counting) performance counters.

The counter-inhibit register mcountinhibit is a **WARL** register that controls which of the hardware performance-monitoring counters increment. The settings in this register only control whether the counters increment; their accessibility is not affected by the setting of this register.

When the CY, IR, or HPM*n* bit in the mcountinhibit register is clear, the mcycle, minstret, or `mhpmcountern` register increments as usual. When the CY, IR, or HPM_n_ bit is set, the corresponding counter does not increment.

The mcycle CSR may be shared between harts on the same core, in which case the mcountinhibit.CY field is also shared between those harts, and so writes to mcountinhibit.CY will be visible to those harts.

If the mcountinhibit register is not implemented, the implementation behaves as though the register were set to zero.

When the mcycle and minstret counters are not needed, it is desirable to conditionally inhibit them to reduce energy consumption. Providing a single CSR to inhibit all counters also allows the counters to be atomically sampled.

Because the `mtime` counter can be shared between multiple cores, it cannot be inhibited with the mcountinhibit mechanism.

C.5.1. Attributes

CSR Address	0x320
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.5.2. Format

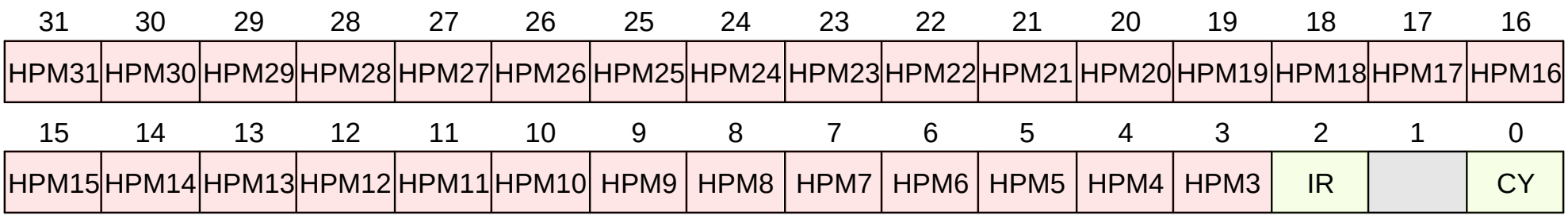


Figure 5. mcountinhibit format

C.5.3. Field Summary

Name	Location	Type	Reset Value
mcountinhibit.CY	0	RW RO	UNDEFINED_LEGAL
mcountinhibit.IR	2	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM3	3	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM4	4	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM5	5	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM6	6	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM7	7	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
mcountinhibit.HPM8	8	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM9	9	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM10	10	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM11	11	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM12	12	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM13	13	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM14	14	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM15	15	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM16	16	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM17	17	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM18	18	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM19	19	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM20	20	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM21	21	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM22	22	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM23	23	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM24	24	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM25	25	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM26	26	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM27	27	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
mcountinhibit.HPM28	28	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM29	29	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM30	30	RW RO	UNDEFINED_LEGAL
mcountinhibit.HPM31	31	RW RO	UNDEFINED_LEGAL

C.5.4. Fields

mcountinhibit.CY Field

Location:

0

Description:

When set, mcycle.COUNT stops counting in all privilege modes.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.IR Field

Location:

2

Description:

When set, minstret.COUNT stops counting in all privilege modes.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM3 Field

Location:

3

Description:

[when="COUNTINHIBIT_EN[3] == true"]
When set, hpmcounter3.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[3] == false"]
Since hpmcounter3 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM4 Field

Location:

4

Description:

[when="COUNTINHIBIT_EN[4] == true"]
When set, hpmcounter4.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[4] == false"]
Since hpmcounter4 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM5 Field

Location:

5

Description:

[when="COUNTINHIBIT_EN[5] == true"]
When set, hpmcounter5.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[5] == false"]
Since hpmcounter5 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM6 Field

Location:

6

Description:

[when="COUNTINHIBIT_EN[6] == true"]
When set, hpmcounter6.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[6] == false"]
Since hpmcounter6 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM7 Field

Location:

7

Description:

[when="COUNTINHIBIT_EN[7] == true"]
When set, hpmcounter7.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[7] == false"]
Since hpmcounter7 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM8 Field

Location:

8

Description:

[when="COUNTINHIBIT_EN[8] == true"]
When set, hpmcounter8.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[8] == false"]
Since hpmcounter8 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM9 Field

Location:

9

Description:

[when="COUNTINHIBIT_EN[9] == true"]
When set, hpmcounter9.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[9] == false"]
Since hpmcounter9 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM10 Field

Location:

10

Description:

[when="COUNTINHIBIT_EN[10] == true"]
When set, hpmcounter10.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[10] == false"]
Since hpmcounter10 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM11 Field

Location:

11

Description:

[when="COUNTINHIBIT_EN[11] == true"]
When set, hpmcounter11.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[11] == false"]
Since hpmcounter11 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM12 Field

Location:

12

Description:

[when="COUNTINHIBIT_EN[12] == true"]
When set, hpmcounter12.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[12] == false"]
Since hpmcounter12 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM13 Field

Location:

13

Description:

[when="COUNTINHIBIT_EN[13] == true"]
When set, hpmcounter13.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[13] == false"]

Since hpmcounter13 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM14 Field

Location:

14

Description:

[when="COUNTINHIBIT_EN[14] == true"]
When set, hpmcounter14.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[14] == false"]
Since hpmcounter14 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM15 Field

Location:

15

Description:

[when="COUNTINHIBIT_EN[15] == true"]
When set, hpmcounter15.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[15] == false"]
Since hpmcounter15 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM16 Field

Location:

16

Description:

[when="COUNTINHIBIT_EN[16] == true"]
When set, hpmcounter16.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[16] == false"]
Since hpmcounter16 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM17 Field

Location:

17

Description:

[when="COUNTINHIBIT_EN[17] == true"]
When set, hpmcounter17.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[17] == false"]
Since hpmcounter17 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM18 Field

Location:

18

Description:

[when="COUNTINHIBIT_EN[18] == true"]
When set, hpmcounter18.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[18] == false"]
Since hpmcounter18 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM19 Field

Location:

19

Description:

[when="COUNTINHIBIT_EN[19] == true"]
When set, hpmcounter19.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[19] == false"]
Since hpmcounter19 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM20 Field

Location:

20

Description:

[when="COUNTINHIBIT_EN[20] == true"]
When set, hpmcounter20.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[20] == false"]
Since hpmcounter20 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM21 Field

Location:

21

Description:

[when="COUNTINHIBIT_EN[21] == true"]
When set, hpmcounter21.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[21] == false"]
Since hpmcounter21 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM22 Field

Location:

22

Description:

[when="COUNTINHIBIT_EN[22] == true"]
When set, hpmcounter22.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[22] == false"]
Since hpmcounter22 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM23 Field

Location:

23

Description:

[when="COUNTINHIBIT_EN[23] == true"]
When set, hpmcounter23.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[23] == false"]
Since hpmcounter23 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM24 Field

Location:

24

Description:

[when="COUNTINHIBIT_EN[24] == true"]
When set, hpmcounter24.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[24] == false"]
Since hpmcounter24 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM25 Field

Location:

25

Description:

[when="COUNTINHIBIT_EN[25] == true"]
When set, hpmcounter25.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[25] == false"]
Since hpmcounter25 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM26 Field

Location:

26

Description:

[when="COUNTINHIBIT_EN[26] == true"]
When set, hpmcounter26.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[26] == false"]

Since hpmcounter26 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM27 Field

Location:

27

Description:

[when="COUNTINHIBIT_EN[27] == true"]
When set, hpmcounter27.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[27] == false"]
Since hpmcounter27 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM28 Field

Location:

28

Description:

[when="COUNTINHIBIT_EN[28] == true"]
When set, hpmcounter28.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[28] == false"]
Since hpmcounter28 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM29 Field

Location:

29

Description:

[when="COUNTINHIBIT_EN[29] == true"]
When set, hpmcounter29.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[29] == false"]
Since hpmcounter29 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM30 Field

Location:

30

Description:

[when="COUNTINHIBIT_EN[30] == true"]
When set, hpmcounter30.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[30] == false"]
Since hpmcounter30 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcountinhibit.HPM31 Field

Location:

31

Description:

[when="COUNTINHIBIT_EN[31] == true"]
When set, hpmcounter31.COUNT stops counting in all privilege modes.

[when="COUNTINHIBIT_EN[31] == false"]
Since hpmcounter31 is not implemented, this field is read-only zero.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.6. 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.6.1. Attributes

CSR Address	0xb00
Defining extension	Sm
Length	64-bit
Privilege Mode	M

C.6.2. Format

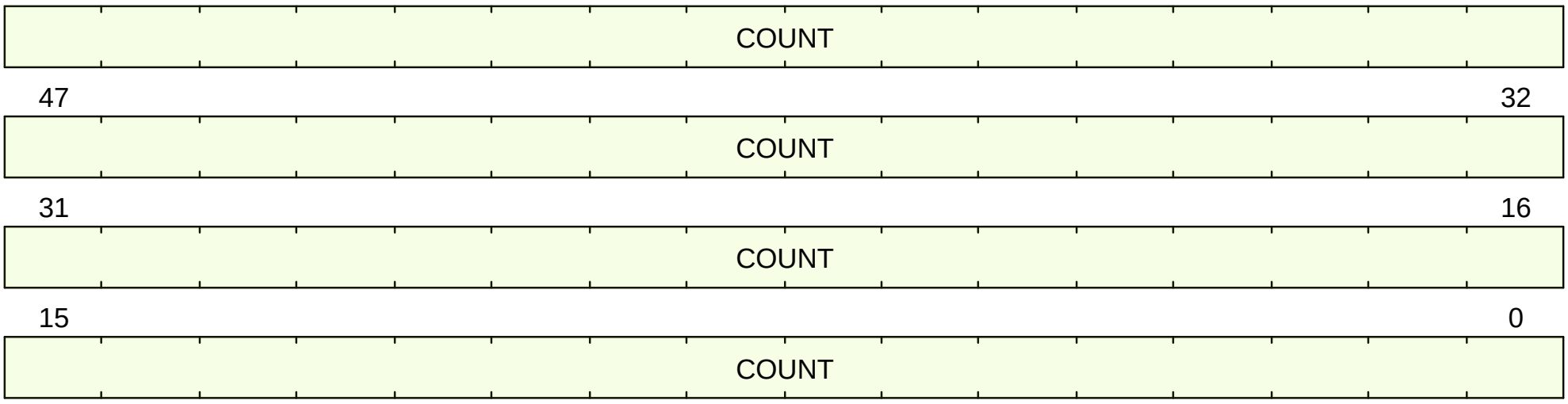


Figure 6. mcycle format

C.6.3. Field Summary

Name	Location	Type	Reset Value
mcycle.COUNT	63:0	RW-RH	UNDEFINED_LEGAL

C.6.4. 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.6.5. 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.6.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return read_mcycle();
```

C.7. mepc

Machine Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in M-mode.

Also controls where the hart jumps on an exception return from M-mode.

C.7.1. Attributes

CSR Address	0x341
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.7.2. Format

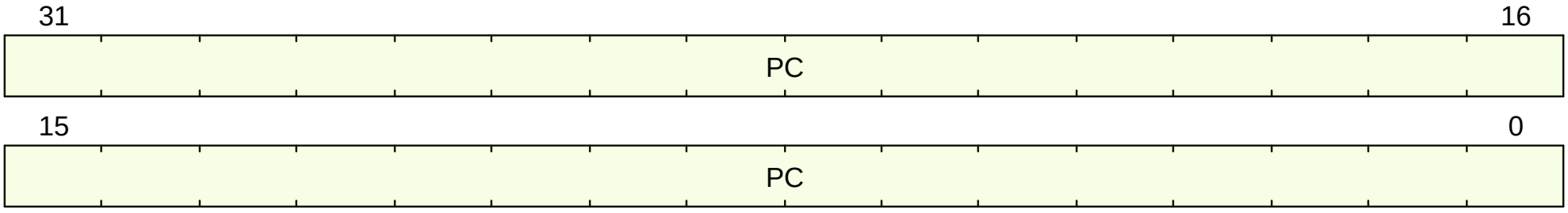


Figure 7. mepc format

C.7.3. Field Summary

Name	Location	Type	Reset Value
mepc.PC	31:0	RW-RH	0

C.7.4. Fields

mepc.PC Field

Location:

31:0

Description:

When a trap is taken into M-mode, mepc.PC is written with the virtual address of the instruction that was interrupted or that encountered the exception.
Otherwise, mepc.PC is never written by the implementation, though it may be explicitly written by software.

On an exception return from M-mode (from the MRET instruction), control transfers to the virtual address read out of mepc.PC.

[when,"ext?(:C)"]
Because PCs are always halfword-aligned, bit 0 of mepc.PC is always read-only 0.

[when,"!ext?(:C)"]
Because PCs are always word-aligned, bits 1:0 of mepc.PC are always read-only 0.

[when,"ext?(:C) && MUTABLE_MISA_C == true"]
When misa.C is clear, bit 1 is masked to zero. Writes to bit 1 are still captured, and may be visible on the next read with misa.C is set.

Type:

RW-RH

Reset value:

0

C.7.5. 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:

```
PC = return csr_value.PC & ~64'b1;
```

C.7.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
if (implemented?(ExtensionName::C) && CSR[misa].C == 1'b1) {
  return CSR[mepc].PC & ~64'b1;
} else {
  return CSR[mepc].PC;
}
```

C.8. mhartid

Machine Hart ID

Reports the unique hart-specific ID in the system.

C.8.1. Attributes

CSR Address	0xf14
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.8.2. Format

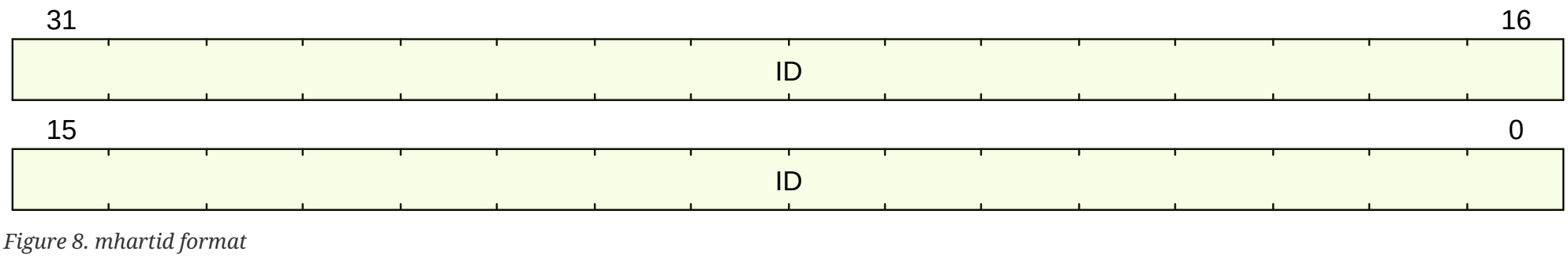


Figure 8. mhartid format

C.8.3. Field Summary

Name	Location	Type	Reset Value
mhartid.ID	31:0	RO	UNDEFINED_LEGAL

C.8.4. Fields

mhartid.ID Field

Location:
31:0

Description:
hart-specific ID.

Type:
RO

Reset value:
UNDEFINED_LEGAL

C.8.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return hartid();
```

C.9. mie

Machine Interrupt Enable

mip.yaml#/description

C.9.1. Attributes

CSR Address	0x304
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.9.2. Format

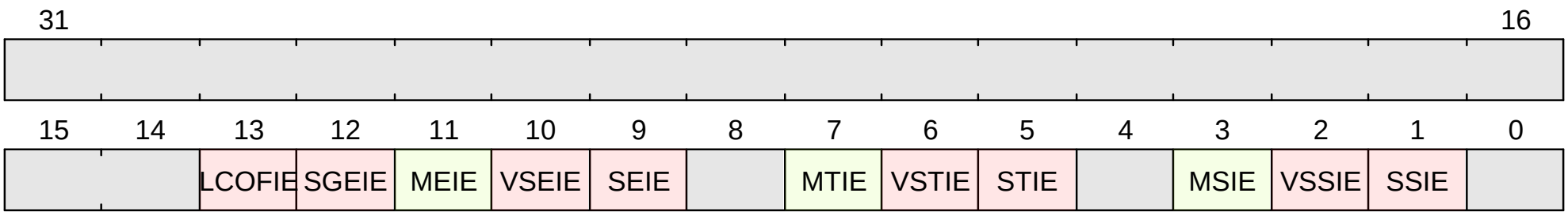


Figure 9. mie format

C.9.3. Field Summary

Name	Location	Type	Reset Value
mie.SSIE	1	RW	0
mie.VSSIE	2	RW	0
mie.MSIE	3	RW	0
mie.STIE	5	RW	0
mie.VSTIE	6	RW	0
mie.MTIE	7	RW	0
mie.SEIE	9	RW	0
mie.VSEIE	10	RW	0
mie.MEIE	11	RW	0
mie.SGEIE	12	RW	0
mie.LCOFIE	13	RW	0

C.9.4. Fields

mie.SSIE Field

Location:

1

Description:

Enables Supervisor Software Interrupts.

Alias of `sie.SSIE` when mideleg.SSI is set. Otherwise, `sie.SSIE` is read-only 0.

Type:

RW

Reset value:

0

mie.VSSIE Field

Location:

2

Description:

Enables Virtual Supervisor Software Interrupts.

Alias of `hie.VSSIE`.

Alias of `vsie.SSIE` when `hideleg.VSSI` is set. Otherwise, `vseie.SSIE` is read-only 0.

Alias of `sie.SSIE` when `hideleg.VSSI` is set and the current mode is VS or VU
(Because mie is inaccessible in VS or VU mode, this alias can never be observed by software).

Type:

RW

Reset value:

0

mie.MSIE Field

Location:

3

Description:

Enables Machine Software Interrupts.

Type:

RW

Reset value:

0

mie.STIE Field

Location:

5

Description:

Enables Supervisor Timer Interrupts.

Alias of sip when mideleg.STI is set. Otherwise, sip is read-only 0.

Type:

RW

Reset value:

0

mie.VSTIE Field

Location:

6

Description:

Enables Virtual Supervisor Timer Interrupts.

Alias of `hie.VSTIE`.

Alias of `vsie.STIE` when `hideleg.VSTI` is set. Otherwise, `vseie.STIE` is read-only 0.

Alias of `sie.STIE` when `hideleg.VSTI` is set and the current mode is VS or VU
(Because mie is inaccessible in VS or VU mode, this alias can never be observed by software).

Type:

RW

Reset value:

0

mie.MTIE Field

Location:

7

Description:

Enables Machine Timer Interrupts.

Type:

RW

Reset value:

0

mie.SEIE Field

Location:

9

Description:

Enables Supervisor External Interrupts.

Alias of `sie.SEIE` when mideleg.SEI is set. Otherwise, `sie.SEIE` is read-only 0.

Type:

RW

Reset value:

0

mie.VSEIE Field

Location:

10

Description:

Enables Virtual Supervisor External Interrupts.

Alias of `hie.VSEIE`.

Alias of `vsie.SEIE` when `hideleg.VSEI` is set. Otherwise, `vseie.SEIE` is read-only 0.

Alias of `sie.SEIE` when `hideleg.VSEI` is set and the current mode is VS or VU
(Because mie is inaccessible in VS or VU mode, this alias can never be observed by software).

Type:

RW

Reset value:

0

mie.MEIE Field

Location:

11

Description:

Enables Machine External Interrupts.

Type:

RW

Reset value:

0

mie.SGEIE Field

Location:

12

Description:

Enables Supervisor Guest External Interrupts

Alias of `hie.SGEIE`.

Type:

RW

Reset value:

0

mie.LCOFIE Field

Location:

13

Description:

Enables Local Counter Overflow Interrupts.

Alias of `sie.LCOFIE` when mideleg.LCOFI is set. Otherwise, `sie.LCOFIE` is an independent writable bit when `mvien.LCOFI` is set or is read-only 0.

Alias of `vsip.LCOFIE` when `hideleg.LCOFI` is set. Otherwise, `vsip.LCOFIE` is read-only 0.

Type:

RW

Reset value:

0

C.10. mimpid

Machine Implementation ID

Reports the vendor-specific implementation ID.

The mimpid CSR provides a unique encoding of the version of the processor implementation. This register must be readable in any implementation, but a value of 0 can be returned to indicate that the field is not implemented. The Implementation value should reflect the design of the RISC-V processor itself and not any surrounding system.

The format of this field is left to the provider of the architecture source code, but will often be printed by standard tools as a hexadecimal string without any leading or trailing zeros, so the Implementation value can be left-justified (i.e., filled in from most-significant nibble down) with subfields aligned on nibble boundaries to ease human readability.

C.10.1. Attributes

CSR Address	0xf13
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.10.2. Format

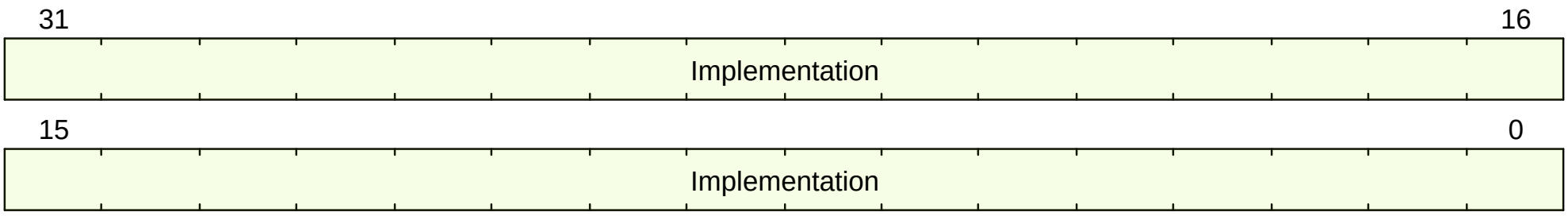


Figure 10. mimpid format

C.10.3. Field Summary

Name	Location	Type	Reset Value
mimpid.Implementation	31:0	RO	UNDEFINED_LEGAL

C.10.4. Fields

mimpid.Implementation Field

Location:

31:0

Description:

Vendor-specific implementation ID.

Type:

RO

Reset value:

UNDEFINED_LEGAL

C.11. 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.11.1. Attributes

CSR Address	0xb02
Defining extension	Sm
Length	64-bit
Privilege Mode	M

C.11.2. Format

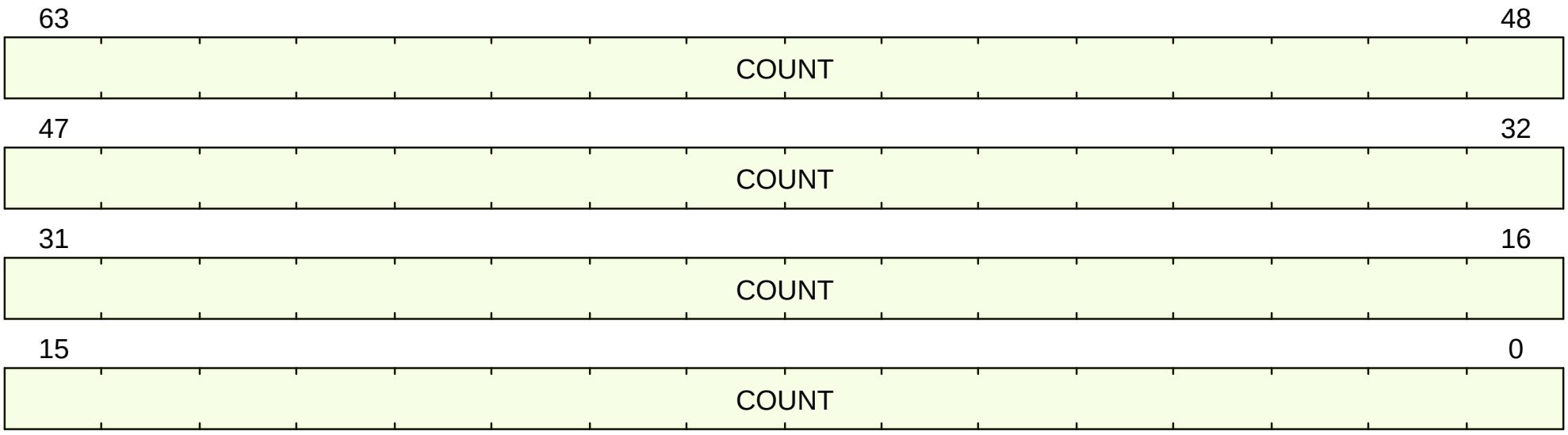


Figure 11. minstret format

C.11.3. Field Summary

Name	Location	Type	Reset Value
minstret.COUNT	63:0	RW-H	UNDEFINED_LEGAL

C.11.4. 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.12. mip

Machine Interrupt Pending

The mie and mip CSRs are MXLEN-bit read/write registers used when the CLINT or PLIC interrupt controllers are present. Note that the CLINT refers to an interrupt controller used by some RISC-V implementations but isn't a ratified RISC-V International standard.

The mip CSR contains information on pending interrupts, while mie is the corresponding CSR containing interrupt enable bits. Interrupt cause number *i* (as reported in the mcause CSR) corresponds to bit *i* in both mip and mie. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform use.

Interrupts designated for platform use may be designated for custom use at the platform's discretion.

An interrupt *i* will trap to M-mode (causing the privilege mode to change to M-mode) if all of the following are true:

- either the current privilege mode is M and the MIE bit in the mstatus register is set, or the current privilege mode has less privilege than M-mode;
- bit *i* is set in both mip and mie
- if register mideleg exists, bit *i* is not set in mideleg.

These conditions for an interrupt trap to occur must be evaluated in a bounded amount of time from when an interrupt becomes, or ceases to be, pending in mip, and must also be evaluated immediately following the execution of an *x*RET instruction or an explicit write to a CSR on which these interrupt trap conditions expressly depend (including mip, mie, mstatus, and mideleg).

Interrupts to M-mode take priority over any interrupts to lower privilege modes.

Each individual bit in register mip may be writable or may be read-only. When bit *i* in mip is writable, a pending interrupt *i* can be cleared by writing 0 to this bit. If interrupt *i* can become pending but bit *i* in mip is read-only, the implementation must provide some other mechanism for clearing the pending interrupt.

A bit in mie must be writable if the corresponding interrupt can ever become pending. Bits of mie that are not writable must be read-only zero.

The machine-level interrupt registers handle a few root interrupt sources which are assigned a fixed service priority for simplicity, while separate external interrupt controllers can implement a more complex prioritization scheme over a much larger set of interrupts that are then muxed into the machine-level interrupt sources.

The non-maskable interrupt is not made visible via the mip register as its presence is implicitly known when executing the NMI trap handler.

If supervisor mode is implemented, bits mip.SEIP and mie.SEIE are the interrupt-pending and interrupt-enable bits for supervisor-level external interrupts. SEIP is writable in mip, and may be written by M-mode software to indicate to S-mode that an external interrupt is pending. Additionally, the platform-level interrupt controller may generate supervisor-level external interrupts. Supervisor-level external interrupts are made pending based on the logical-OR of the software-writable SEIP bit and the signal from the external interrupt controller. When mip is read with a CSR instruction, the value of the SEIP bit returned in the `rd` destination register is the logical-OR of the software-writable bit and the interrupt signal from the interrupt controller, but the signal from the interrupt controller is not used to calculate the value written to SEIP. Only the software-writable SEIP bit participates in the read-modify-write sequence of a CSRRS or CSRRC instruction.

For example, if we name the software-writable SEIP bit B and the signal from the external interrupt controller E, then if `csrrs t0, mip, t1` is executed, `t0[9]` is written with `B || E`, then B is written with `B || t1[9]`. If `csrrw t0, mip, t1` is executed, then `t0[9]` is written with `B || E`, and B is simply written with `t1[9]`. In neither case does B depend upon E.

The SEIP field behavior is designed to allow a higher privilege layer to mimic external interrupts cleanly, without losing any real external interrupts. The behavior of the CSR instructions is slightly modified from regular CSR accesses as a result.

If supervisor mode is implemented, bits mip.STIP and mie.STIE are the interrupt-pending and interrupt-enable bits for supervisor-level timer interrupts. STIP is writable in mip, and may be written by M-mode software to deliver timer interrupts to S-mode.

If supervisor mode is implemented, bits mip.SSIP and mie.SSIE are the interrupt-pending and interrupt-enable bits for supervisor-level software interrupts. SSIP is writable in mip and may also be set to 1 by a platform-specific interrupt controller.

<% if ext?(:Sscofpmf) -%> bits mip.LCOFIP and mie.LCOFIE are the interrupt-pending and interrupt-enable bits for local counter-overflow interrupts. LCOFIP is read-write in mip and reflects the occurrence of a local counter-overflow overflow interrupt request resulting from any of the `mhpmeventn`.OF bits being set. <% end -%>

Multiple simultaneous interrupts destined for M-mode are handled in the following decreasing priority order: MEI, MSI, MTI, SEI, SSI, STI, LCOFI.

The machine-level interrupt fixed-priority ordering rules were developed with the following rationale.

Interrupts for higher privilege modes must be serviced before interrupts for lower privilege modes to support preemption.

The platform-specific machine-level interrupt sources in bits 16 and above have platform-specific priority, but are typically chosen

to have the highest service priority to support very fast local vectored interrupts.

External interrupts are handled before internal (timer/software) interrupts as external interrupts are usually generated by devices that might require low interrupt service times.

Software interrupts are handled before internal timer interrupts, because internal timer interrupts are usually intended for time slicing, where time precision is less important, whereas software interrupts are used for inter-processor messaging. Software interrupts can be avoided when high-precision timing is required, or high-precision timer interrupts can be routed via a different interrupt path. Software interrupts are located in the lowest four bits of mip as these are often written by software, and this position allows the use of a single CSR instruction with a five-bit immediate.

Restricted views of the mip and mie registers appear as the sip and `sie` registers for supervisor level. If an interrupt is delegated to S-mode by setting a bit in the mideleg register, it becomes visible in the sip register and is maskable using the `sie` register. Otherwise, the corresponding bits in sip and `sie` are read-only zero.

C.12.1. Attributes

CSR Address	0x344
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.12.2. Format

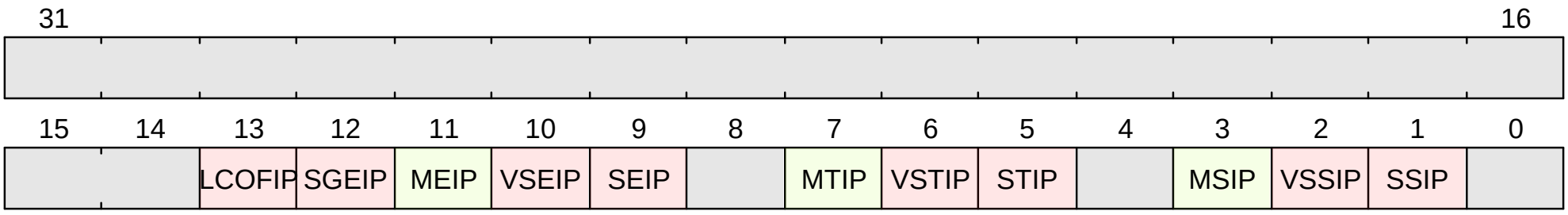


Figure 12. mip format

C.12.3. Field Summary

Name	Location	Type	Reset Value
mip.SSIP	1	RW	0
mip.VSSIP	2	RW	0
mip.MSIP	3	RO	0
mip.STIP	5	RW	0
mip.VSTIP	6	RO-H	0
mip.MTIP	7	RO-H	0
mip.SEIP	9	RW-H	0
mip.VSEIP	10	RO-H	0
mip.MEIP	11	RO-H	0
mip.SGEIP	12	RO-H	0

Name	Location	Type	Reset Value
mip.LCOFIP	13	RW-H	0

C.12.4. Fields

mip.SSIP Field

Location:

1

Description:

Supervisor Software Interrupt Pending

Reports the current pending state of an (H)S-mode software interrupt, which is generated by writing to this field.

<%- if ext?(:Smaia) -%>
When using AIA/IMSIC, IPIs are expected to be delivered as external interrupts
and SSIP is not backed by any hardware update (aside from any aliasing effects).
However, SSIP is still writable by M-mode software and, when written, can be used to
generate an S-mode Software Interrupt.
<%- end -%>

<% if ext?(:Smaia) %>_Aliases_<% else %>_Alias_<% end %>:

- sip.SSIP when mideleg.SSI is set
  <%- if ext?(:Smaia) -%>
- `mvip.SSIP`
  <%- end -%>

Type:

RW

Reset value:

0

mip.VSSIP Field

Location:

2

Description:

Virtual Supervisor Software Interrupt Pending

Reports the current pending state of a VS-mode software interrupt, which is generated by writing to this field.

<%- if ext?(:Smaia) -%>
When using AIA/IMSIC, IPIs are expected to be delivered as external interrupts and VSSIP is not backed by any hardware update (aside from any aliased writes).
However, VSSIP is still writable by M-mode software and, when written, can be used to
generate a VS-mode Software Interrupt.
<%- end -%>

Aliases:

- `hip.VSSIP`
- `hvip.VSSIP`
- `vsip.SSIP` when `hideleg.VSSI` is set

Type:

RW

Reset value:

0

mip.MSIP Field

Location:

3

Description:

Machine Software Interrupt Pending

Unused field.

<%- if ext?(:Smaia) -%>
With AIA/IMSIC, IPIs are delivered as external interrupts. As a result, this bit is unused and hardwired to 0.
<%- end -%>

Type:

RO

Reset value:

0

mip.STIP Field

Location:

5

Description:

Supervisor Timer Interrupt Pending

Reports the current pending state of an (H)S-mode timer interrupt
<%- if ext?(:Sstc) -%>
, which is normally controlled by the `stimecmp` CSR.
<%- else -%>
, which is generated by software by writing to mip.STIP<% if ext?(:Smaia) %>or its alias `mvip.STIP`<% end %>.
<%- end -%>

<%-if ext?(:Sstc) -%>
When menvcfg.STCE is set, mip.STIP is RO-H, and is completely controlled by the timer interrupt device (using `stimecmp`).

When menvcfg.STCE is clear, mip.STIP is RW, and M-mode software may write the bit to inject a Supervisor Timer Interrupt.
<%- end -%>

<% if ext?(:Smaia) %>_Aliases_<% else %>_Alias_<% end %>:

- sip.STIP when mideleg.STI is set (though sip.STIP is a read-only view)
  <%- if ext?(:Smaia) -%>
- `mvip.STIP` when when menvcfg.STCE is clear
  <%- end -%>

Type:

RW

Reset value:

0

mip.VSTIP Field

Location:

6

Description:

Virtual Supervisor Timer Interrupt Pending

Reports the current pending state of a VS-mode timer interrupt
<%- if ext?(:Sstc) -%>
, which is normally controlled by the `vstimecmp` CSR, but can also be injected by the hypervisor through `hvip.VSTIP`.
<%- else -%>
, which is generated by M-mode and/or HS-mode software by writing to `hvip.VSTIP`.
<%- end -%>

<%-if ext?(:Sstc) -%>
When menvcfg.STCE is set (enabling the Sstc extension), mip.VSTIP is the logical OR of `hvip.VSTIP` and the VS-level interrupt signal generated by the timer device (controlled by the value of `vstimecmp`).

When menvcfg.STCE is clear (disabling the Sstc extension), mip.VSTIP is exactly the value of `hvip.VSTIP`.
<%- end -%>

mip.VSTIP is never writable. If VS-mode software wants to clear the bit, it must do so
<%- if ext?(:Sstc) -%>
by writing the `vstimecmp` register or
<%- end -%>
by calling into the hypervisor (which can then clear `hvip.VSTIP`).

Aliases:

- `hip.VSTIP`
- `vsip.STIP` when `hideleg.VSTI` is set
- `hvip.VSTIP` <% if ext?(:Sstc) %>when menvcfg.STCE is clear<% end %> (though `hvip.VSTIP` is writable)

Type:

RO-H

Reset value:

0

mip.MTIP Field

Location:

7

Description:

Machine Timer Interrupt Pending

Reports the current pending state of an M-mode timer interrupt.

Bit is controlled by the timer device (using `mtimecmp`), and is not writable.

Type:

RO-H

Reset value:

0

mip.SEIP Field

Location:

9

Description:

Supervisor External Interrupt Pending

Reports the current pending state of an (H)S-mode external interrupt.

This field has two parts: a software-writable shadow value and a wire from the interrupt controller.
The value presented to software in the bit on a CSR read is the logical OR of the software-writable value and the interrupt controller value.
When software writes this bit, only the shadow value is updated (the interrupt controller is not notified of the write).

<%- if ext?(:Smaia) -%>

The software-writable shadow value is aliased in `mvip.SEIP` (Smaia extension).
<%- end -%>

Alias:

- sip.SEIP when mideleg.SEI is set (though sip.SEIP is read-only)

Type:

RW-H

Reset value:

0

mip.VSEIP Field

Location:

10

Description:

Virtual Supervisor External Interrupt Pending

Reports the current pending state of a VS-mode external interrupt.

This field is the logical OR of `hvip.VSEIP` and the wire coming from the interrupt controller.

The field is not writable by software
<%- if ext?(:Smaia) -%>
(*i.e.*, unlike the behavior of mip.SEIP/`mvip.SEIP`, attempted writes to mip.VSEIP do not propagate to `hvip.VSEIP`)
<%- end -%>

1. +
   Aliases:

   - `hip.VSEIP`
   - `vsip.SEIP` when `hideleg.VSEI` is set

Type:

RO-H

Reset value:

0

mip.MEIP Field

Location:

11

Description:

Machine External Interrupt Pending

Reports the current pending state of an M-mode external interrupt.

MEIP is controlled by the external interrupt controller <% if ext?(:Smaia) %>(AIA) <% end %>.
It is not writable by software.

Type:

RO-H

Reset value:

0

mip.SGEIP Field

Location:

12

Description:

Supervisor Guest External Interrupt Pending

Read-only summary of any pending Supervisor Guest External Interrupt Pending, i.e.:
the logical-OR reduction of the `hgeip` register.

Alias:

- `hip.SGEIP`

Type:

RO-H

Reset value:

0

mip.LCOFIP Field

Location:

13

Description:

Local Counter Overflow Interrupt pending

<%- if ext?(:H) -%>
When `hideleg.LCOFI` is set,
`vsip.LCOFIP`, sip.LCOFIP, and mip.LCOFIP are all aliases.
<%- end -%>

When a counter overflow interrupt occurs, a hidden sticky bit is set.

Software writes 0 to mip.LCOFIP to clear the pending interrupt.

<% if ext?(:H) %>_Aliases_<% else %>_Alias_<% end %>:

- sip.LCOFIP when mideleg.LCOFI is set
  <%- if ext?(:H) -%>
- `vsip.LCOFIP` when `hideleg.LCOFI` is set
  <%- end -%>

Type:

RW-H

Reset value:

0

C.12.5. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return $bits(CSR[CSR[mip]]) | ((CSR[misa].S == 1'b1 && pending_smode_external_interrupt) ? 10'h200 : 0);
```

C.13. misa

Machine ISA Control

Reports the XLEN and "major" extensions supported by the ISA.

C.13.1. Attributes

CSR Address	0x301
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.13.2. Format

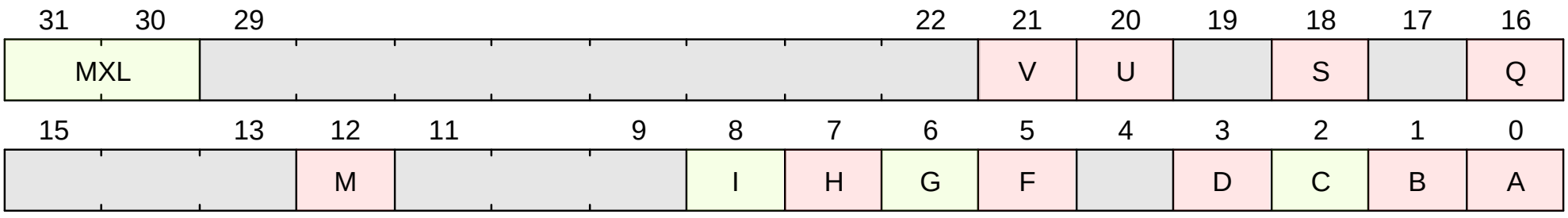


Figure 13. misa format

C.13.3. Field Summary

Name	Location	Type	Reset Value
misa.MXL	31:30	RO	1
misa.A	0	RW RO	UNDEFINED_LEGAL
misa.B	1	RW RO	UNDEFINED_LEGAL
misa.C	2	RW RO	1
misa.D	3	RW RO	UNDEFINED_LEGAL
misa.F	5	RW RO	UNDEFINED_LEGAL
misa.G	6	RO-H RO	UNDEFINED_LEGAL
misa.H	7	RW RO	UNDEFINED_LEGAL
misa.I	8	RO	1
misa.M	12	RW RO	UNDEFINED_LEGAL
misa.Q	16	RW RO	1
misa.S	18	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
misa.U	20	RW RO	UNDEFINED_LEGAL
misa.V	21	RW RO	UNDEFINED_LEGAL

C.13.4. Fields

misa.MXL Field

Location:

31:30

Description:

XLEN in M-mode.

Type:

RO

Reset value:

1

misa.A Field

Location:

0

Description:

Indicates support for the A (atomic) extension.

[when,"MUTABLE_MISA_A == true"]
Writing 0 to this field will cause all atomic instructions to raise an `IllegalInstruction` exception.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.B Field

Location:

1

Description:

Indicates support for the B (bitmanip) extension.

[when,"MUTABLE_MISA_B == true"]
Writing 0 to this field will cause all bitmanip instructions to raise an `IllegalInstruction` exception.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.C Field

Location:

2

Description:

Indicates support for the C (compressed) extension.

[when,"MUTABLE_MISA_C == true"]
Writing 0 to this field will cause all compressed instructions to raise an `IllegalInstruction` exception.
Additionally, IALIGN becomes 32.

Type:

RW

RO

Reset value:

1

misa.D Field

Location:

3

Description:

Indicates support for the D (double precision float) extension.

[when,"MUTABLE_MISA_D == true"] +—+ Writing 0 to this field will cause all double-precision floating point instructions to raise an `IllegalInstruction` exception.

Additionally, the upper 32-bits of the f registers will read as zero. +—+

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.F Field

Location:

5

Description:

Indicates support for the F (single precision float) extension.

[when,"MUTABLE_MISA_F == true"] +—+ Writing 0 to this field will cause all floating point (single and double precision) instructions to raise an `IllegalInstruction` exception.

Writing 0 to this field with misa.D set will result in UNDEFINED behavior. +—+

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.G Field

Location:

6

Description:

Indicates support for all of the following extensions: I, A, M, F, D.

Type:

RO-H

RO

Reset value:

UNDEFINED_LEGAL

misa.H Field

Location:

7

Description:

Indicates support for the H (hypervisor) extension.

[when,"MUTABLE_MISA_H == true"]
Writing 0 to this field will cause all attempts to enter VS- or VU- mode, execute a hypervisor instruction, or access a hypervisor CSR to raise an `IllegalInstruction` fault.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.I Field

Location:

8

Description:

Indicates support for the I (base) extension.

Type:

RO

Reset value:

1

misa.M Field

Location:

12

Description:

Indicates support for the M (integer multiply/divide) extension.

[when,"MUTABLE_MISA_M == true"]
Writing 0 to this field will cause all attempts to execute an integer multiply or divide instruction to raise an `IllegalInstruction` exception.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.Q Field

Location:

16

Description:

Indicates support for the Q (quad precision float) extension.

[when,"MUTABLE_MISA_Q == true"] + — + Writing 0 to this field will cause all quad-precision floating point instructions to raise an `IllegalInstruction` exception. + — +

Type:

RW

RO

Reset value:

1

misa.S Field

Location:

18

Description:

Indicates support for the S (supervisor mode) extension.

[when,"MUTABLE_MISA_S == true"]
Writing 0 to this field will cause all attempts to enter S-mode or access S-mode state to raise an exception.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.U Field

Location:

20

Description:

Indicates support for the U (user mode) extension.

[when,"MUTABLE_MISA_U == true"]
Writing 0 to this field will cause all attempts to enter U-mode to raise an exception.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

misa.V Field

Location:

21

Description:

Indicates support for the V (vector) extension.

[when,"MUTABLE_MISA_V == true"]
Writing 0 to this field will cause all attempts to execute a vector instruction to raise an `IllegalInstruction` trap.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.13.5. 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:

```
MXL = csr_value.MXL
A = csr_value.A
B = csr_value.B
C = csr_value.C
D = csr_value.D
F = if (csr_value.F == 0 && csr_value.D == 1) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}

# fall-through; write the intended value
return csr_value.F;

G = csr_value.G
H = csr_value.H
I = csr_value.I
M = csr_value.M
Q = if ((csr_value.F == 0 || csr_value.D == 0) && csr_value.Q == 1) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}

# fall-through; write the intended value
return csr_value.Q;

S = csr_value.S
U = csr_value.U
V = csr_value.V
```

C.13.6. Software read

This CSR may return a value that is different from what is stored in hardware.

```
return CSR[misa].MXL `<< (xlen() - 2 | (CSR[misa].V `<< 21) | (CSR[misa].U `<< 20) | (CSR[misa].S `<< 18) | (CSR[misa].Q `<< 16) |
(CSR[misa].M `<< 12) | (CSR[misa].I `<< 7) | (CSR[misa].H `<< 6) | ((CSR[misa].A & CSR[misa].M & CSR[misa].F & CSR[misa].D) `<< 5)
| (CSR[misa].F `<< 4) | (CSR[misa].D `<< 3) | (CSR[misa].C `<< 2) | (CSR[misa].B `<< 1) | CSR[misa].A);
```

C.14. mscratch

Machine Scratch Register

Scratch register for software use. Bits are not interpreted by hardware.

C.14.1. Attributes

CSR Address	0x340
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.14.2. Format

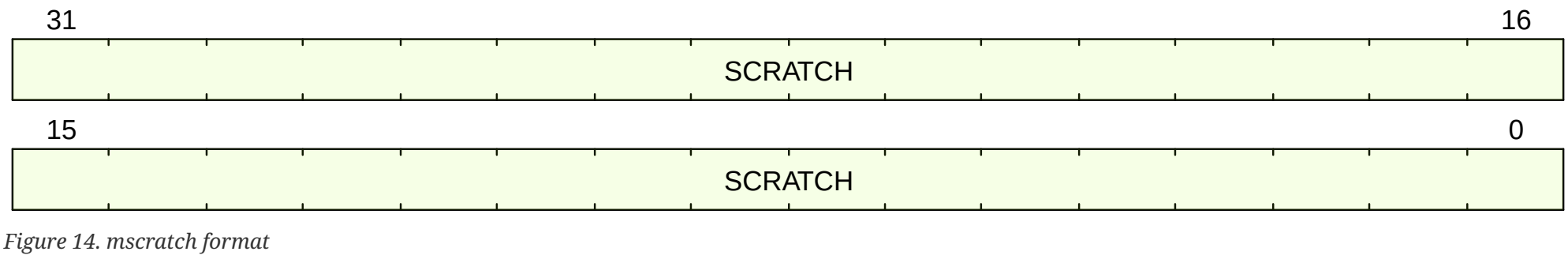


Figure 14. mscratch format

C.14.3. Field Summary

Name	Location	Type	Reset Value
mscratch.SCRATCH	31:0	RW	0

C.14.4. Fields

mscratch.SCRATCH Field

Location:

31:0

Description:

Scratch value

Type:

RW

Reset value:

0

C.15. mstatus

Machine Status

The mstatus register tracks and controls the hart's current operating state.

C.15.1. Attributes

CSR Address	0x300
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.15.2. Format

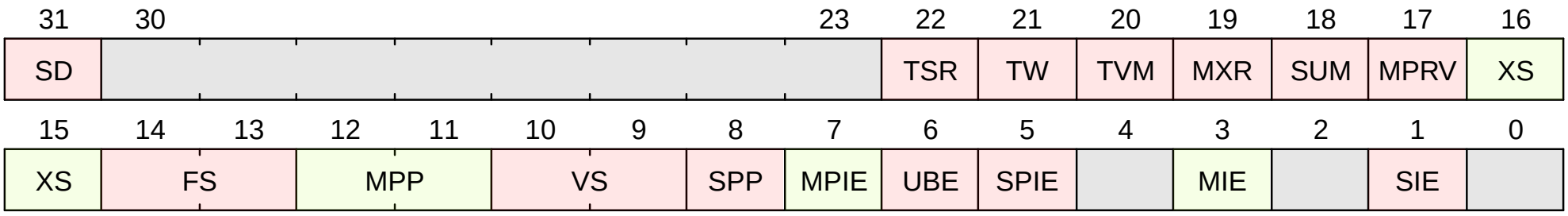


Figure 15. mstatus format

C.15.3. Field Summary

Name	Location	Type	Reset Value
mstatus.SD	31	RO-H RO-H RO	UNDEFINED_LEGAL
mstatus.TSR	22	RW	UNDEFINED_LEGAL
mstatus.TW	21	RW	UNDEFINED_LEGAL
mstatus.TVM	20	RO RW	UNDEFINED_LEGAL
mstatus.MXR	19	RW	UNDEFINED_LEGAL
mstatus.SUM	18	RW RO	UNDEFINED_LEGAL
mstatus.MPRV	17	RW-H RO	0
mstatus.XS	16:15	RO	0
mstatus.FS	14:13	RW-H RO RO RW	UNDEFINED_LEGAL
mstatus.MPP	12:11	RW-H	3

Name	Location	Type	Reset Value
mstatus.VS	10:9	RW-H RO RO RW	UNDEFINED_LEGAL
mstatus.SPP	8	RW-H	UNDEFINED_LEGAL
mstatus.MPIE	7	RW-H	UNDEFINED_LEGAL
mstatus.UBE	6	RW RO	UNDEFINED_LEGAL
mstatus.SPIE	5	RW-H RO	UNDEFINED_LEGAL
mstatus.MIE	3	RW-H	0
mstatus.SIE	1	RW-H RO	UNDEFINED_LEGAL

C.15.4. Fields

mstatus.SD Field

Location:

31

Description:

Read-only bit that summarizes whether either the FS, XS, or VS fields signal the presence of some dirty state.

Type:

RO-H

RO-H

RO

Reset value:

UNDEFINED_LEGAL

mstatus.TSR Field

Location:

22

Description:

When 1, attempts to execute the sret instruction while executing in HS/S-mode will raise an Illegal Instruction exception.

[when,"ext?(:H)"]
Does not affect the behavior of sret in VS_mode (see hstatus.VTSR).

Type:

RW

Reset value:

UNDEFINED_LEGAL

mstatus.TW Field

Location:

21

Description:

When 1, the WFI instruction will raise an Illegal Instruction trap after an implementaion-defined wait period when executed in a mode other than M-mode.

When 0, the wfi instruction is permitted to wait forever in (H)S-mode but must trap after an implementation-defined wait period in U-mode.

Type:

RW

Reset value:

UNDEFINED_LEGAL

mstatus.TVM Field

Location:

20

Description:

When 1, an `Illegal Instruction` trap occurs when

- writing the satp CSR, executing an sfence.vma, or executing an sinval.vma while in (H)S-mode (but not VS-mode)
- writing the `hgtap` CSR, executing an hfence.gvma, or executing an hinval.gvma while in HS-mode

  Notably, mstatus.TVM does **not** cause

  *hfence.vvma, sfence.w.inval, or sfence.inval.ir to trap.
- Any additional traps in VS-mode (controlled via hstatus.VTVM instead).

Type:

RO

RW

Reset value:

UNDEFINED_LEGAL

mstatus.MXR Field

Location:

19

Description:

When 1, loads from pages marked readable **or executable** are allowed.
When 0, loads from pages marked executable raise a Page Fault exception.

Type:

RW

Reset value:

UNDEFINED_LEGAL

mstatus.SUM Field

Location:

18

Description:

When 0, an S-mode read or an M-mode read with mstatus.MPRV=1 and mstatus.MPP=01
to a 'U' (user) page will cause an ILLEGAL INSTRUCTION exception.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.MPRV Field

Location:

17

Description:

When 1, loads and stores behave as if the current virtualization mode:privilege level was
mstatus.MPV:mstatus.MPP.

mstatus.MPRV is cleared on any exception return (mret or sret instruction, regardless of the trap handler privilege mode).

Type:

RW-H

RO

Reset value:

0

mstatus.XS Field

Location:

16:15

Description:

Summarizes the current state of any custom extension state.
Either 0 - Off, 1 - Initial, 2 - Clean, 3 - Dirty.
Since there are no custom extensions in the base spec, this field is read-only 0.

Type:

RO

Reset value:

0

mstatus.FS Field

Location:

14:13

Description:

When 0, floating point instructions (from F and D extensions) are disabled,
and cause `ILLEGAL INSTRUCTION` exceptions.
When a floating point register, or the fCSR register is written, FS obtains the value 3.
Values 1 and 2 are valid write values for software, but are not interpreted by hardware
other than to possibly enable a previously-disabled floating point unit.

Type:

RW-H

RO

RO

RW

Reset value:

UNDEFINED_LEGAL

mstatus.MPP Field

Location:

12:11

Description:

Written by hardware in two cases:

- Written with the prior nominal privilege level when entering M-mode from an exception/interrupt.
- Written with 0 when executing an mret instruction to return from an exception in M-mode.

  Can also be written by software without immediate side-effect.

  Affects execution in two cases:

- On a return from an exception from M-mode, the machine will
  enter the privilege level stored in MPP before clearing the field.
- When mstatus.MPRV is set, loads and stores behave as if the current privilege level were MPP.

Type:

RW-H

Reset value:

3

mstatus.VS Field

Location:

10:9

Description:

When 0, vector instructions (from the V extension) are disabled, and cause ILLEGAL INSTRUCTION exceptions.
When a vector register or vector CSR is written, VS obtains the value 3.
Values 1 and 2 are valid write values for software, but are not interpreted by hardware
other than to possibly enable a previously-disabled vector unit.

Type:

RW-H

RO

RO

RW

Reset value:

UNDEFINED_LEGAL

mstatus.SPP Field

Location:

8

Description:

Written by hardware in two cases:

- Written with the prior nominal privilege level when entering (H)S-mode from an exception/interrupt.
- Written with 0 when executing an sret instruction to return from an exception in (H)S-mode or (unlikely) M-mode.

  Can also be written by software without immediate side-effect.

  Affects execution in one case:

- On a return from an exception using the sret instruction in (H)S-mode or (unlikely) M-mode,
  the machine will enter the privilege level stored in SPP before clearing the field.

  Notably, mstatus.SPP does not affect exception return in VS-mode (see vsstatus.SPP).

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

mstatus.MPIE Field

Location:

7

Description:

Written by hardware in two cases:

- Written with prior value of mstatus.MIE when entering M-mode from an exception/interrupt.
- Written with the value 1 when returning from an exception in M-mode (via the mret instruction).

  Can also be written by software without immediate side effect.

  Other than serving as a record of nested traps as described above, mstatus.MPIE does not affect execution.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

mstatus.UBE Field

Location:

6

Description:

Controls the endianness of U-mode (0 = little, 1 = big).
Instructions are always little endian, regardless of the data setting.

[when,"U_MODE_ENDIANNESS == 'little'"]
Since the CPU does not support big endian in U-mode, this is hardwired to 0.

[when,"U_MODE_ENDIANNESS == 'big'"]
Since the CPU does not support little endian in U-mode, this is hardwired to 1.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.SPIE Field

Location:

5

Description:

Written by hardware in two cases:

- Written with prior value of mstatus.SIE when entering (H)S-mode from an exception/interrupt.
- Written with the value 1 when returning from an exception via the sret instruction in (H)S-mode or (unlikely) M-mode.

Can also be written by software without immediate side effect.

Other than serving as a record of nested traps as described above, mstatus.SPIE does not affect execution.

Type:

RW-H

RO

Reset value:

UNDEFINED_LEGAL

mstatus.MIE Field

Location:

3

Description:

Written by hardware in two cases:

- Written with the value 0 when entering M-mode from an exception/interrupt.
- Written with the prior value of mstatus.MPIE when returning from an exception in M-mode (via mret).

Affects execution by:

- When 0, all interrupts are disabled when the current privilege level is M.
- When 1, interrupts that are not otherwise disabled with a field in mie are enabled.

Type:

RW-H

Reset value:

0

mstatus.SIE Field

Location:

1

Description:

Written by hardware in two cases:

- Written with the value 0 when entering (H)S-mode from an exception/interrupt.
- Written with the prior value of mstatus.SPIE when returning from an exception via sret in (H)S-mode or (unlikely) M-mode.

Affects execution by:

- When 0, all (H)S-mode interrupts are disabled when the current privilege level is (H)S (M-mode interrupts are still enabled).
- When 1, (H)S-mode interrupts that are not otherwise disabled with a field in `sie` are enabled.

Type:

RW-H

RO

Reset value:

UNDEFINED_LEGAL

C.15.5. 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:

```
SD = csr_value.SD
TSR = csr_value.TSR
TW = csr_value.TW
TVM = if (CSR[misa].S == 1'b0) {
  return 0;
} else if (MSTATUS_TVM_IMPLEMENTED) {
  return csr_value.TVM;
} else {
  return 0;
}

MXR = csr_value.MXR
SUM = csr_value.SUM
MPRV = csr_value.MPRV
XS = csr_value.XS
FS = if (MISA_CSR_IMPLEMENTED && (CSR[misa].S == 1'b0) && (CSR[misa].F == 1'b0)) {
  # must be read-only-0
  return 0;
}
return $array_includes?(MSTATUS_FS_LEGAL_VALUES, csr_value.FS) ? csr_value.FS : UNDEFINED_LEGAL_DETERMINISTIC;

MPP = if (csr_value.MPP == 2'b01 && !implemented?(ExtensionName::S)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (csr_value.MPP == 2'b00 && !implemented?(ExtensionName::U)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (csr_value.MPP == 2'b10) {
  # never a valid value
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else {
  return csr_value.MPP;
}

VS = if (implemented?(ExtensionName::V) && CSR[misa].V == 1'b1){
  return $array_includes?(MSTATUS_VS_LEGAL_VALUES, csr_value.VS) ? csr_value.VS : UNDEFINED_LEGAL_DETERMINISTIC;
} else if (!implemented?(ExtensionName::S) && !implemented?(ExtensionName::V)) {
  # must be read-only-0
  return 0;
} else {
  # there will be no hardware update in this case because we know the V extension isn't implemented
  return $array_includes?(MSTATUS_VS_LEGAL_VALUES, csr_value.VS) ? csr_value.VS : UNDEFINED_LEGAL_DETERMINISTIC;
}

SPP = if (csr_value.SPP == 2'b10) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else {
  return csr_value.SPP;
}

MPIE = csr_value.MPIE
UBE = csr_value.UBE
SPIE = csr_value.SPIE
MIE = csr_value.MIE
SIE = csr_value.SIE
```

C.16. mtval

Machine Trap Value

Holds trap-specific information

C.16.1. Attributes

CSR Address	0x343
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.16.2. Format

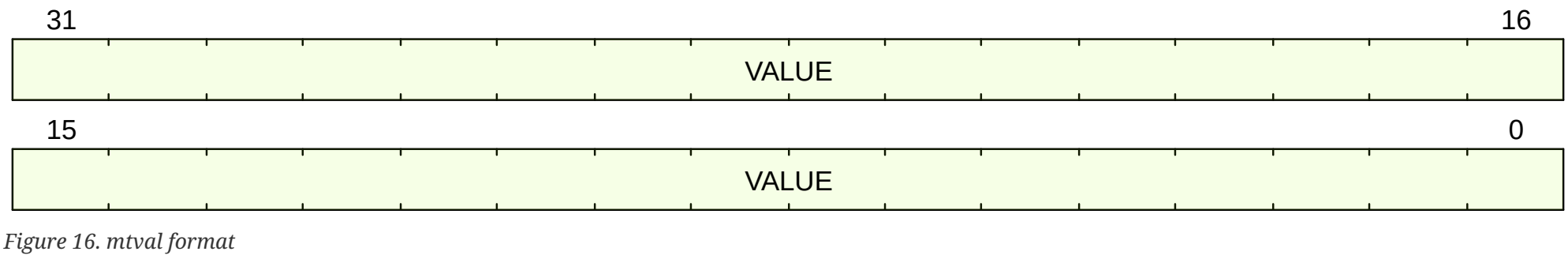


Figure 16. mtval format

C.16.3. Field Summary

Name	Location	Type	Reset Value
mtval.VALUE	31:0	RW-H	0

C.16.4. Fields

mtval.VALUE Field

Location:

31:0

Description:

Written with trap-specific information when a trap is taken into M-mode.

The values are:

[separator="!"]
!===
! Exception type ! Value

! [0] Instruction address misaligned ! The misaligned virtual PC (same as the value written to mepc).
! [1] Instruction access fault ! The <% if ext?(:C) %> portion of the <% end %> virtual PC causing the access fault <%- unless ext?(:C) -%>(same as the value written to mepc)<%- end -%>.
! [2] Illegal Instruction ! The encoding of the illegal instruction.
! [3] Breakpoint
! [when,"REPORT_VA_IN_MTVAL_ON_BREAKPOINT == true"]
When caused by an EBREAK instruction, the virtual PC of the breakpoint instruction.

[when,"REPORT_VA_IN_MTVAL_ON_BREAKPOINT == false"]
When caused by an EBREAK instruction, zero.

When caused by a data address (*i.e.*, watchpoint) breakpoint, the faulting virtual address.
When caused by an instruction address breakpoint, the faulting virtual PC.
! [4] Load address misaligned ! The misaligned virtual load address.
! [5] Load access fault
! The part of virtual load address causing in the access fault.

When the load is misaligned, the reported value is the smallest address on the page causing a fault

(*e.g.*, if an 8-byte load is equally split across a page and the fault occurs on the second page,
address + 4 is reported).

(Even though the access fault arises on a physical address, the virtual address is reported)
! [6] Store/AMO address misaligned ! The misaligned virtual store/AMO address.
! [7] Store/AMO access fault
! The virtual store/AMO address causing the access fault.

When the store/AMO is misaligned, the reported value is the smallest address on the page causing a fault
(*e.g.*, if an 8-byte store is equally split across a page and the fault occurs on the second page,
address + 4 is reported).

(Even though the access fault arises on a physical address, the virtual address is reported)
! [8] Environment call from U-mode <% if ext?(:H) %>or VU-mode<% end %> ! Zero
! [9] Environment call from (H)S-mode ! Zero
<%- if ext?(:H) -%>
! [10] Environment call from VS-mode ! Zero
<%- end -%>
! [11] Environment call from M-mode ! Zero
! [12] Instruction page fault
! The <% if ext?(:C) %> portion of the <% end %> virtual PC causing the page fault
<% unless ext?(:C) %>(same as the value written to mepc)<% end %>.
! [13] Load page fault
! The part of the virtual load address causing in the page fault.

When the load is misaligned, the reported value is the smallest address on the page causing a fault
(*e.g.*, if an 8-byte load is equally split across a page and the fault occurs on the second page, address + 4 is reported).
! [15] Store/AMO page fault
! The virtual store/AMO address causing in the page fault.

When the store/AMO is misaligned, the reported value is the smallest address on the page causing a fault
(*e.g.*, if an 8-byte store is equally split across a page and the fault occurs on the second page, address + 4 is reported).
<%- if ext?(:H) -%>
! [20] Instruction guest-page fault
! The <% if ext?(:C) %> portion of the <% end %> virtual PC causing the fault <% unless ext?(:C) %>(same as the value written to mepc)<% end %>.

The guest physical address is reported in mtval2.
! [21] Load guest-page fault
! The part of the virtual address causing the fault.

When the load is misaligned, the reported value is the smallest address on the page causing a fault
(*e.g.*, if an 8-byte load is equally split across a page and the fault occurs on the second page, address + 4 is reported).

The guest physical address is reported in mtval2.
! [22] Virtual instruction
! The encoding of the faulting virtual instruction.
! [23] Store/AMO guest-page fault
! The part of the virtual address causing the fault.

When the store/AMO is misaligned, the reported value is the smallest address on the page causing a fault
(*e.g.*, if an 8-byte store is equally split across a page and the fault occurs on the second page, address + 4 is reported).

The guest physical address is reported in mtval2.
<%- end -%>
!===

Type:

RW-H

Reset value:

0

C.17. mtvec

Machine Trap Vector Control

Controls where traps jump.

C.17.1. Attributes

CSR Address	0x305
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.17.2. Format

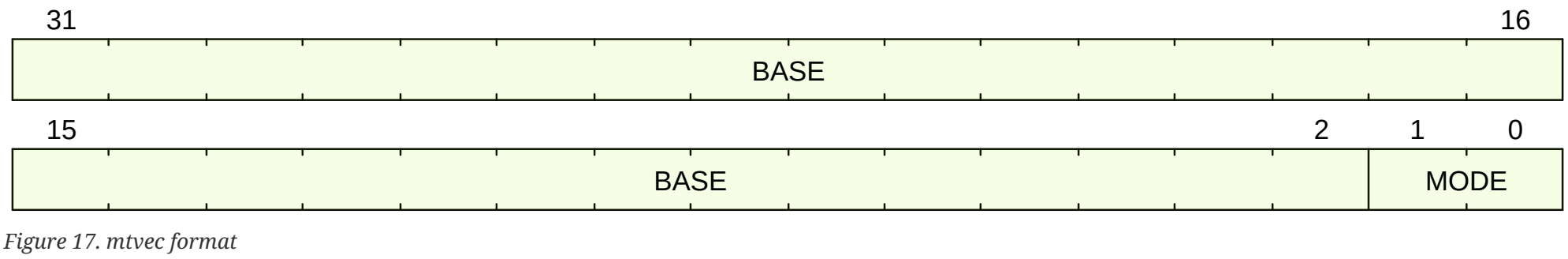


Figure 17. mtvec format

C.17.3. Field Summary

Name	Location	Type	Reset Value
mtvec.BASE	31:2	RO RW-R	0
mtvec.MODE	1:0	RO RO RW-R	UNDEFINED_LEGAL

C.17.4. Fields

mtvec.BASE Field

Location:

31:2

Description:

Bits [MXLEN-1:2] of the exception vector physical address for any trap taken in M-mode.

The implementation physical memory map may resitrict which values are legal in this field.

Type:

RO

RW-R

Reset value:

0

mtvec.MODE Field

Location:

1:0

Description:

Vectoring mode for asynchronous interrupts.

0 - Direct, 1 - Vectored

When Direct, all synchronous exceptions and asynchronous interrupts jump to (mtvec.BASE << 2).

When Vectored, asynchronous interrupts jump to (mtvec.BASE << 2 + mcause*4) while synchronous exceptions continue to jump to (mtvec.BASE << 2).

Type:

RO

RO

RW-R

Reset value:

UNDEFINED_LEGAL

C.17.5. 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:

```
BASE = # Base spec says that BASE must be 4-byte aligned, which will always be the case
# implementations may put further constraints on BASE when MODE != Direct
# If that is the case, stvec should have an override for the implementation

if (csr_value.MODE == 0) {
  if ($array_includes?(MTVEC_MODES, 0)) {
    return csr_value.BASE;
  }
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "retain") {
    return CSR[mtvec].BASE;
  }
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "custom") {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
  unreachable();
} else if (csr_value.MODE == 1) {
  if ($array_includes?(MTVEC_MODES, 1)) {
    return csr_value.BASE;
  }
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "retain") {
    return CSR[mtvec].BASE;
  }
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "custom") {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
  unreachable();
} else {
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "retain") {
    return CSR[mtvec].BASE;
  }
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "custom") {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
  unreachable();
}

MODE = if (csr_value.MODE == 0) {
  if ($array_includes?(MTVEC_MODES, 0)) {
    return csr_value.MODE;
  } else {
    if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "retain") {
      return CSR[mtvec].MODE;
    } else if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "custom") {
      return UNDEFINED_LEGAL_DETERMINISTIC;
    }
  }
} else if (csr_value.MODE == 1) {
  if ($array_includes?(MTVEC_MODES, 1)) {
    return csr_value.MODE;
```

```
  } else {
    if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "retain") {
      return CSR[mtvec].MODE;
    } else if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "custom") {
      return UNDEFINED_LEGAL_DETERMINISTIC;
    }
  }
} else {
  if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "retain") {
    return CSR[mtvec].MODE;
  } else if (MTVEC_ILLEGAL_WRITE_BEHAVIOR == "custom") {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}
unreachable();
```

C.18. mvendorid

Machine Vendor ID

Reports the JEDEC manufacturer ID of the core.

C.18.1. Attributes

CSR Address	0xf11
Defining extension	Sm
Length	32-bit
Privilege Mode	M

C.18.2. Format

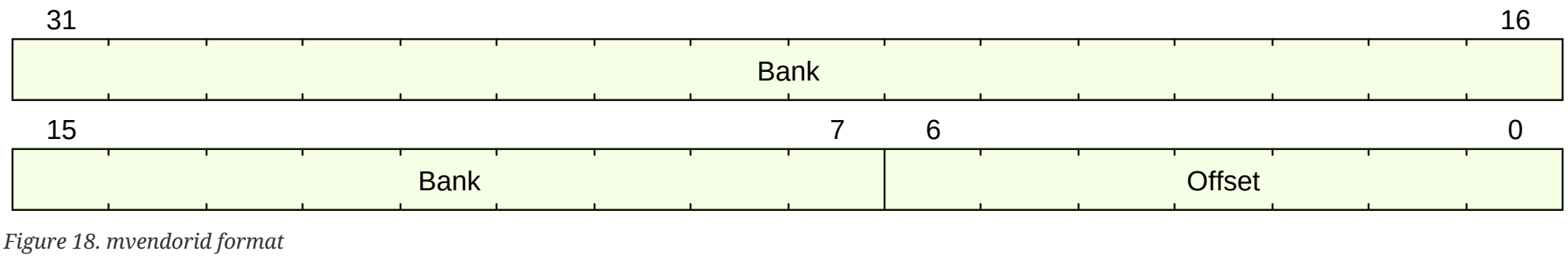


Figure 18. mvendorid format

C.18.3. Field Summary

Name	Location	Type	Reset Value
mvendorid.Bank	31:7	RO	UNDEFINED_LEGAL
mvendorid.Offset	6:0	RO	UNDEFINED_LEGAL

C.18.4. Fields

mvendorid.Bank Field

Location:
31:7

Description:
JEDEC manufacturer ID bank minus 1

Type:
RO

Reset value:
UNDEFINED_LEGAL

mvendorid.Offset Field

Location:
6:0

Description:
JEDEC manufacturer ID offset

Type:
RO

Reset value:
UNDEFINED_LEGAL

C.19. 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.19.1. Attributes

CSR Address	0xc01
Defining extension	Zicntr
Length	64-bit
Privilege Mode	U

C.19.2. Format

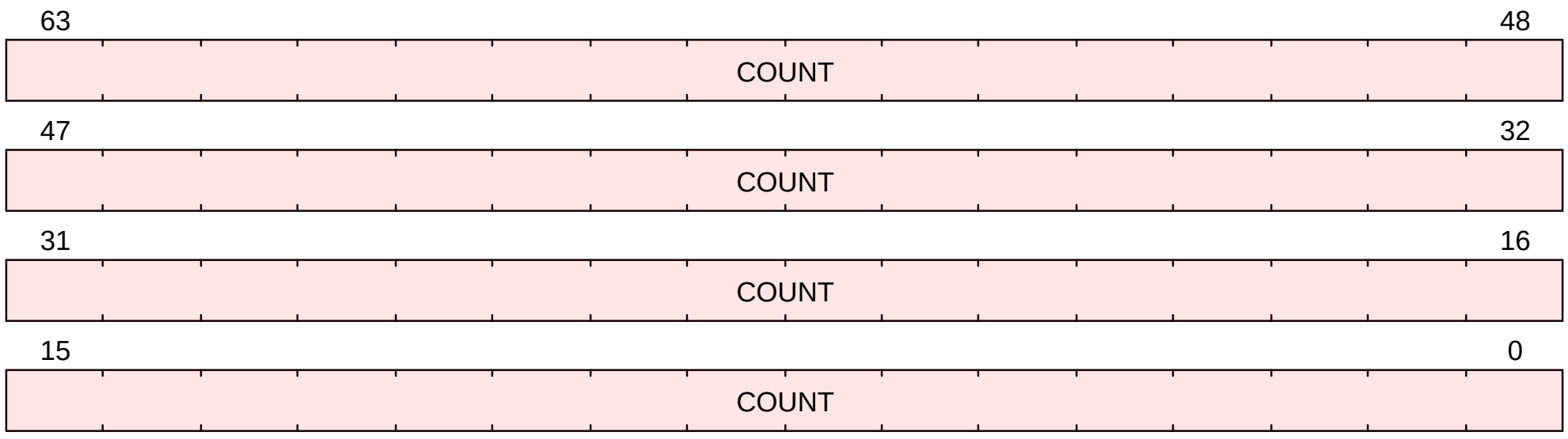


Figure 19. time format

C.19.3. Field Summary

Name	Location	Type	Reset Value
time.COUNT	63:0	RO-H	UNDEFINED_LEGAL

C.19.4. 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.19.5. 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 (CSR[mcounteren].TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].TM & CSR[scounteren].TM) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].TM == 1'b0 && CSR[mcounteren].TM == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].TM & CSR[scounteren].TM) == 1'b0) && (CSR[mcounteren].IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mtime();
```

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. unreachable (builtin)

Indicate that the IDL line should be unreachable.

If this function is called, it represents a bug in the IDL code.

Return Type	`void`
Arguments	None

D.11. 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.12. 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.13. read_mcycle (builtin)

Return the current value of the cycle counter.

Return Type	`Bits`
Arguments	None

D.14. read_mtime (builtin)

Return the current value of the real time device.

Return Type	`Bits`
Arguments	None

D.15. 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.16. 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.17. 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.18. 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.19. 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.20. 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.21. 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.22. 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.23. 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.24. 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.25. 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.26. abort_current_instruction (builtin)

Abort the current instruction, and start refetching from $pc.

Return Type	`void`
Arguments	None

D.27. ebreak (builtin)

Raise an `Environment Break` exception, returning control to the debug environment.

Return Type	`void`
Arguments	None

D.28. prefetch_instruction (builtin)

Hint to prefetch a block containing virtual_address for an upcoming fetch.

Return Type	`void`
Arguments	`XReg virtual_address`

D.29. prefetch_read (builtin)

Hint to prefetch a block containing virtual_address for an upcoming load.

Return Type	`void`
Arguments	`XReg virtual_address`

D.30. prefetch_write (builtin)

Hint to prefetch a block containing virtual_address for an upcoming store.

Return Type	`void`
Arguments	`XReg virtual_address`

D.31. 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.32. fence_tso (builtin)

Execute a TSO memory ordering fence.(according to the FENCE instruction).

Return Type	`void`
Arguments	None

D.33. ifence (builtin)

Execute a memory ordering instruction fence (according to FENCE.I).

Return Type	`void`
Arguments	None

D.34. 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.35. pow (generated)

Return `value` to the power `exponent`.

Return Type	`XReg`
Arguments	`XReg value, XReg exponent`

D.36. 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.37. 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.38. 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.39. 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.40. 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.41. 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.42. read_physical_memory_8 (builtin)

Read a byte from physical memory.

Return Type	`Bits⑧`
Arguments	`XReg paddr`

D.43. read_physical_memory_16 (builtin)

Read two bytes from physical memory.

Return Type	`Bits⑯`
Arguments	`XReg paddr`

D.44. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

D.45. read_physical_memory_64 (builtin)

Read eight bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

D.46. 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.47. write_physical_memory_8 (builtin)

Write a byte to physical memory.

Return Type	`void`

Arguments	`XReg paddr, Bits<8> value`

D.48. write_physical_memory_16 (builtin)

Write two bytes to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<16> value`

D.49. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<32> value`

D.50. write_physical_memory_64 (builtin)

Write eight bytes to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<64> value`

D.51. 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.52. 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.53. 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.54. 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.55. 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.56. 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.57. 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.58. 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.59. set_external_interrupt

Set an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  CSR[mip].MEIP = 1'b1;
} else if ((CSR[misa].S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  pending_smode_external_interrupt = true;
} else if ((CSR[misa].H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  CSR[mip].VSEIP = 1'b1;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.60. clear_external_interrupt

Clear an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  CSR[mip].MEIP = 1'b0;
```

```
} else if ((CSR[misa].S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  pending_smode_external_interrupt = false;
} else if ((CSR[misa].H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  CSR[mip].VSEIP = 1'b0;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.61. set_software_interrupt

Set a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  CSR[mip].MSIP = 1'b1;
} else if ((CSR[misa].S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  CSR[mip].SSIP = 1'b1;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.62. clear_software_interrupt

Clear a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

```
if (target_mode == PrivilegeMode::M) {
  CSR[mip].MSIP = 1'b0;
} else if ((CSR[misa].S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  CSR[mip].SSIP = 1'b0;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.63. 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) {
  CSR[mip].MTIP = 1'b1;
} else if ((CSR[misa].S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  CSR[mip].STIP = 1'b1;
} else if ((CSR[misa].H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  pending_vsmode_timer_interrupt = true;
} else {
  assert(false, "Invalid target_mode");
}
```

```
refresh_pending_interrupts();
```

D.64. 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) {
  CSR[mip].MTIP = 1'b0;
} else if ((CSR[misa].S == 1'b1) && (target_mode == PrivilegeMode::S)) {
  CSR[mip].STIP = 1'b0;
} else if ((CSR[misa].H == 1'b1) && (target_mode == PrivilegeMode::VS)) {
  pending_vsmode_timer_interrupt = false;
} else {
  assert(false, "Invalid target_mode");
}
refresh_pending_interrupts();
```

D.65. 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 = CSR[CSR[mip]].sw_read() & $bits(CSR[CSR[mie]]);
if (pending_ints == 0) {
  pending_and_enabled_interrupts = 0;
  return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") &&
implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_enabled_ints = mode() == PrivilegeMode::M) && (CSR[mstatus].MIE == 1'b0 ? 0 : ($bits(CSR[CSR[mie]]) &
(HAS_MIDELEG ? ~$bits(CSR[CSR[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 (CSR[misa].S == 1'b1) {
  Bits<MXLEN> smode_enabled_ints = mode() == PrivilegeMode::M) || (CSR[mstatus].SIE == 1'b0 ? 0 : $bits(CSR[CSR[mie]]) &
($bits(CSR[CSR[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.66. 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[$bits(InterruptCode::MachineTimer)] == 1'b1) {
  return InterruptCode::MachineTimer;
} else if (CSR[misa].S == 1'b1) {
  if (int_mask[$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;
  }
} else if (implemented?(ExtensionName::Sscofpmf)) {
  if (int_mask[$bits(InterruptCode::LocalCounterOverflow)] == 1'b1) {
    return InterruptCode::LocalCounterOverflow;
  }
}
assert(false, "There is no valid interrupt");
```

D.67. 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(CSR[CSR[mideleg]]) : MXLEN'0);
if (mmode_pending_and_enabled != 0) {
  assert((mode() != PrivilegeMode::M) || (CSR[mstatus].MIE == 1'b1), "M-mode interrupts are not enabled");
  chosen = highest_priority_interrupt(mmode_pending_and_enabled);
} else if (CSR[misa].S == 1'b1) {
  Bits<MXLEN> smode_pending_and_enabled = (pending_and_enabled_interrupts & $bits(CSR[CSR[mideleg]]));
  if (smode_pending_and_enabled != 0) {
    assert((mode() == PrivilegeMode::U) || (mode() == PrivilegeMode::VU) || (mode() == PrivilegeMode::VS) || (mode() ==
PrivilegeMode::S) && (CSR[mstatus].SIE == 1'b1), "S-mode interrupt can't be triggered");
    chosen = highest_priority_interrupt(smode_pending_and_enabled);
  }
}
assert($bits(chosen) != 0, "Didn't pick interrupt?");
PrivilegeMode to_mode;
Bits<MXLEN> chosen_mask = (MXLEN'1 << $bits(chosen));
if (((HAS_MIDELEG ? $bits(CSR[CSR[mideleg]]) : MXLEN'0) & chosen_mask) == 0) {
  to_mode = PrivilegeMode::M;
} else {
  if (CSR[misa].S == 1'b1) {
    to_mode = PrivilegeMode::S;
  } else {
    to_mode = PrivilegeMode::U;
  }
}
return chosen, to_mode;
```

D.68. 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) {
  CSR[mepc].PC = $pc;
  CSR[mstatus].MPP = $bits(mode())[1:0];
  if (CSR[misa].H == 1'b1) {
    if (MXLEN == 64) {
      CSR[mstatus].MPV = $bits(mode())[2];
    } else {
      CSR[mstatush].MPV = $bits(mode())[2];
    }
    CSR[mtval2].VALUE = 0;
    CSR[mtinst].VALUE = 0;
  }
  CSR[mcause].CODE = $bits(code);
  CSR[mcause].INT = 1'b1;
  CSR[mtval].VALUE = 0;
  if (CSR[mtvec].MODE == 0) {
    $pc = {CSR[mtvec].BASE, 2'b00};
  } else if (CSR[mtvec].MODE == 1'b1) {
    $pc = {CSR[mtvec].BASE, 2'b00} + ($bits(code) * 4);
  }
} else if ((CSR[misa].S == 1'b1) && (to_mode == PrivilegeMode::S)) {
  CSR[sepc].PC = $pc;
  CSR[mstatus].SPP = $bits(mode())[0];
  if (CSR[misa].H == 1'b1) {
    CSR[hstatus].SPV = $bits(mode())[2];
  }
  CSR[scause].CODE = $bits(code);
  CSR[scause].INT = 1'b1;
  CSR[stval].VALUE = 0;
  if (CSR[stvec].MODE == 0) {
    $pc = {CSR[stvec].BASE, 2'b00};
  } else if (CSR[stvec].MODE == 1'b1) {
    $pc = {CSR[stvec].BASE, 2'b00} + ($bits(code) * 4);
  }
} else if ((CSR[misa].H == 1'b1) && (to_mode == PrivilegeMode::VS)) {
  CSR[vsepc].PC = $pc;
  CSR[vsstatus].SPP = $bits(mode())[0];
  CSR[vscause].CODE = $bits(code);
  CSR[vscause].INT = 1'b1;
  CSR[vstval].VALUE = 0;
  if (CSR[vstvec].MODE == 0) {
    $pc = {CSR[vstvec].BASE, 2'b00};
  } else if (CSR[vstvec].MODE == 1'b1) {
    $pc = {CSR[vstvec].BASE, 2'b00} + ($bits(code) * 4);
  }
}
set_mode_no_refresh(to_mode);
```

D.69. 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 (CSR[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.70. 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 (CSR[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.71. 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.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. max_va_size

Returns the largest possible Virtual Address width in any supported translation mode.

The max VA is determined by physical address size when in M mode or S-mode with Bare translation. Otherwise, max VA is the size of a virtual address in the largest supported Sv* mode.

Return the largest that applies.

Return Type	`Bits⑧`
Arguments	None

```
Bits<8> translated_va_size = 0;
if (implemented?(ExtensionName::Sv57)) {
  translated_va_size = 57;
} else if (implemented?(ExtensionName::Sv48)) {
  translated_va_size = 48;
} else if (implemented?(ExtensionName::Sv39)) {
  translated_va_size = 39;
```

```
} else if (implemented?(ExtensionName::Sv32)) {
  translated_va_size = 32;
}
if (PHYS_ADDR_WIDTH > translated_va_size) {
  if (PHYS_ADDR_WIDTH > MXLEN) {
    return MXLEN;
  } else {
    return PHYS_ADDR_WIDTH;
  }
} else {
  return translated_va_size;
}
```

D.74. 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.75. 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.76. 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.77. 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.78. 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.79. 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.80. 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.81. 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.82. set_fp_flag

Add flag to the sticky flags bits in CSR[fcsr]

Return Type	`void`
Arguments	`FpFlag flag`

```
if (flag == FpFlag::NX) {
  CSR[fcsr].NX = 1;
} else if (flag == FpFlag::UF) {
  CSR[fcsr].UF = 1;
} else if (flag == FpFlag::OF) {
  CSR[fcsr].OF = 1;
} else if (flag == FpFlag::DZ) {
  CSR[fcsr].DZ = 1;
} else if (flag == FpFlag::NV) {
  CSR[fcsr].NV = 1;
}
```

D.83. 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, CSR[fcsr].FRM);
} else {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

D.84. 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") {
  CSR[mstatus].FS = 3;
} else if (HW_MSTATUS_FS_DIRTY_UPDATE == "imprecise") {
  unpredictable("The hart may or may not update mstatus.FS now");
}
```

D.85. 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.86. 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 && CSR[misa].F == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
if (CSR[mstatus].FS == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

D.87. 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.88. 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.89. 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.90. 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.91. 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.92. 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.93. 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.94. 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.95. 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.96. 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.97. 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.98. 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.99. 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.100. 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.101. 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.102. 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.103. 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.104. 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.105. 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.106. 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.107. signF32UI

Extract sign-bit of a 32-bit floating point number

Return Type	`Bits①`
Arguments	`Bits<32> a`

```
return a[31];
```

D.108. expF32UI

Extract exponent of a 32-bit floating point number

Return Type	`Bits⑧`
Arguments	`Bits<32> a`

```
return a[30:23];
```

D.109. fracF32UI

Extract significand of a 32-bit floating point number

Return Type	`Bits`
Arguments	`Bits<32> a`

```
return a[22:0];
```

D.110. 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.111. 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.112. 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.113. 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.114. 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.115. 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.116. 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.117. 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.118. 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 & 0x7FFFFFF) == 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.119. 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.120. 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.121. 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.122. 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.123. 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.124. 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.125. 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.126. 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.127. 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.128. vector_state

Get the current vector state from CSRs

Return Type	`VectorState`
Arguments	None

```
VectorState state;
state.log2_sew = 3 + CSR[vtype].VSEW;
state.sew = 7'b1 << state.log2_sew;
Bits<3> vlmul = CSR[vtype].VLMUL;
state.lmul_type = CSR[vtype].VLMUL[2] == 1'b1 ? VectorLmulType::Divide : VectorLmulType::Multiply;
state.log2_lmul = CSR[vtype].VLMUL[1:0];
if (vlmul == 3'b101) {
  state.log2_lmul = 3;
} else if (vlmul == 3'b110) {
  state.log2_lmul = 2;
} else if (vlmul == 3'b111) {
  state.log2_lmul = 1;
} else if (vlmul == 3'b100) {
  unpredictable("VLMUL value 0b100 is reserved");
}
return state;
```

D.129. 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.130. 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.131. 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.132. 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.133. 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(CSR[CSR[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(CSR[CSR[medeleg]]) & (MXLEN'1 << $bits(exception_code))) != 0) {
    if (($bits(CSR[CSR[hedeleg]]) & (MXLEN'1 << $bits(exception_code))) != 0) {
      return PrivilegeMode::VS;
    } else {
      return PrivilegeMode::HS;
    }
  } else {
    return PrivilegeMode::M;
  }
}
```

D.134. creg2reg

Maps a C register index (*e.g.*, `rs1'` in the specification) to an X register index. From the specification:

Table 9. 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.135. 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.136. 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.137. 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.138. 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.139. 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.140. 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.141. 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.142. 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) {
  CSR[htval].VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  CSR[htinst].VALUE = tinst_value;
  CSR[sepc].PC = $pc;
  if (!stval_readonly?()) {
    CSR[stval].VALUE = stval_for(code, gva);
  }
  $pc = {CSR[stvec].BASE, 2'b00};
```

```
  CSR[scause].INT = 1'b0;
  CSR[scause].CODE = $bits(code);
  CSR[hstatus].GVA = 1;
  CSR[hstatus].SPV = 1;
  CSR[hstatus].SPVP = $bits(from_mode)[0];
  CSR[mstatus].SPP = $bits(from_mode)[0];
} else {
  assert(handling_mode == PrivilegeMode::M, "unexpected privilege mode");
  CSR[mtval2].VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  CSR[mtinst].VALUE = tinst_value;
  CSR[mstatus].MPP = $bits(from_mode)[1:0];
  if (MXLEN == 64) {
    CSR[mstatus].MPV = 1;
  } else {
    CSR[mstatush].MPV = 1;
  }
}
set_mode(handling_mode);
abort_current_instruction();
```

D.143. 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.144. 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) {
  CSR[mepc].PC = $pc;
  if (!mtval_readonly?()) {
    CSR[mtval].VALUE = mtval_for(exception_code, tval);
  }
  $pc = {CSR[mtvec].BASE, 2'b00};
  CSR[mcause].INT = 1'b0;
  CSR[mcause].CODE = $bits(exception_code);
  if (CSR[misa].H == 1) {
    CSR[mtval2].VALUE = 0;
    CSR[mtinst].VALUE = 0;
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      if (MXLEN == 32) {
        CSR[mstatush].MPV = 1;
      } else {
        CSR[mstatus].MPV = 1;
      }
    } else {
      if (MXLEN == 32) {
```

```
        CSR[mstatush].MPV = 0;
      } else {
        CSR[mstatus].MPV = 0;
      }
    }
  }
  CSR[mstatus].MPP = $bits(from_mode);
} else if (CSR[misa].S == 1 && (handling_mode == PrivilegeMode::S)) {
  CSR[sepc].PC = $pc;
  if (!stval_readonly?()) {
    CSR[stval].VALUE = stval_for(exception_code, tval);
  }
  $pc = {CSR[stvec].BASE, 2'b00};
  CSR[scause].INT = 1'b0;
  CSR[scause].CODE = $bits(exception_code);
  CSR[mstatus].SPP = $bits(from_mode)[0];
  if (CSR[misa].H == 1) {
    CSR[htval].VALUE = 0;
    CSR[htinst].VALUE = 0;
    CSR[hstatus].SPV = $bits(from_mode)[2];
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      CSR[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) {
        CSR[hstatus].GVA = 1;
      } else {
        CSR[hstatus].GVA = 0;
      }
      CSR[hstatus].SPVP = $bits(from_mode)[0];
    } else {
      CSR[hstatus].SPV = 0;
      CSR[hstatus].GVA = 0;
    }
  }
} else if (CSR[misa].H == 1 && (handling_mode == PrivilegeMode::VS)) {
  CSR[vsepc].PC = $pc;
  if (!vstval_readonly?()) {
    CSR[vstval].VALUE = vstval_for(exception_code, tval);
  }
  $pc = {CSR[vstvec].BASE, 2'b00};
  CSR[vscause].INT = 1'b0;
  CSR[vscause].CODE = $bits(exception_code);
  CSR[vsstatus].SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();
```

D.145. ialign

Returns IALIGN, the smallest instruction encoding size, in bits.

Return Type	Bits⑥
Arguments	None

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 0x1)) {
  return 16;
} else {
  return 32;
}
```

D.146. 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.147. 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.148. valid_interrupt_code?

Returns true if *code* is a legal interrupt number.

Return Type	`Boolean`
Arguments	`XReg code`

```
if (code > 1 `<< $enum_element_size(InterruptCode - 1)) {
  return false;
}
if ($array_includes?($enum_to_a(InterruptCode), code)) {
  return true;
} else {
  return false;
}
```

D.149. valid_exception_code?

Returns true if *code* is a legal exception number.

Return Type	`Boolean`

Arguments	XReg code

```
if (code > 1 `<< $enum_element_size(ExceptionCode - 1)) {
  return false;
}
if ($array_includes?($enum_to_a(ExceptionCode), code)) {
  return true;
} else {
  return false;
}
```

D.150. 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 (CSR[misa].MXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[misa].MXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    } else {
      unreachable();
    }
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    if (CSR[mstatus].SXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[mstatus].SXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    } else {
      unreachable();
    }
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    if (CSR[mstatus].UXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[mstatus].UXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    } else {
      unreachable();
    }
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    if (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    } else {
      unreachable();
    }
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU) {
    if (CSR[vsstatus].UXL == $bits(XRegWidth::XLEN32)) {
      return 32;
    } else if (CSR[vsstatus].UXL == $bits(XRegWidth::XLEN64)) {
      return 64;
    } else {
      unreachable();
    }
  }
}
```

D.151. 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.152. mask_eaddr

Mask upper N bits of an effective address if pointer masking is enabled

Return Type	`XReg`
Arguments	`XReg eaddr`

```
return eaddr;
```

D.153. 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_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 - 1:0];
    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.154. 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.155. 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.156. 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) ? CSR[mstatush].MPV : CSR[mstatus].MPV;
} else {
  assert(false, "TODO");
}
```

D.157. 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 (CSR[misa].U == 1 && CSR[mstatus].MPRV == 1) {
    if (CSR[mstatus].MPP == 0b00) {
      if (CSR[misa].H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VU;
      } else {
        return PrivilegeMode::U;
      }
    } else if (CSR[misa].S == 1 && CSR[mstatus].MPP == 0b01) {
      if (CSR[misa].H == 1 && mpv() == 0b1) {
        return PrivilegeMode::VS;
      } else {
        return PrivilegeMode::S;
      }
    }
  }
}
return mode();
```

D.158. 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.159. 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.160. 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 CSR[misa].MXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    return CSR[mstatus].SXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    return CSR[mstatus].UXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    return CSR[hstatus].VSXL == $bits(xlen32);
  } else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU, "Unexpected mode");
    return CSR[vsstatus].UXL == $bits(xlen32);
  }
}
```

D.161. base64?

return True iff current effective XLEN == 64

Return Type	`Boolean`
Arguments	None

```
return xlen() == 64;
```

D.162. 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 (CSR[misa].H == 1'b1) {
  if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU) {
    Bits<4> mode_val = CSR[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) && (CSR[hstatus].VSXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
        if ((effective_mode == PrivilegeMode::VU) && (CSR[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 && CSR[hstatus].VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[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 && CSR[hstatus].VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[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 && CSR[hstatus].VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[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 (CSR[misa].S == 1'b1) {
  assert(effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U, "unexpected priv mode");
  Bits<4> mode_val = CSR[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 && CSR[mstatus].SXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::U && CSR[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 && CSR[mstatus].SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[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 && CSR[mstatus].SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[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 && CSR[mstatus].SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[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.163. 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, CSR[hgatp].MODE);
```

D.164. 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 = CSR[mstatus].MXR == 1;
if (GSTAGE_MODE_BARE && CSR[hgatp].MODE == $bits(HgatpMode::Bare)) {
  result.paddr = gpaddr;
  return result;
} else if (SV32X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv32x4)) {
  return gstage_page_walk<32, 34, 32, 2>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV39X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv39x4)) {
  return gstage_page_walk<39, 56, 64, 3>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV48X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv48x4)) {
  return gstage_page_walk<48, 56, 64, 4>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV57X4_TRANSLATION && CSR[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.165. 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 (($array_size(VSXLEN) == 1 && VSXLEN[0] == 32) || MXLEN == 64) && (CSR[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 (($array_size(VSXLEN) == 1 && VSXLEN[0] == 32) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002020;
      } else {
        return 0x00003020;
      }
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  }
} else {
  if (REPORT_GPA_IN_TVAL_ON_INTERMEDIATE_GUEST_PAGE_FAULT) {
    if (($array_size(VSXLEN) == 1 && VSXLEN[0] == 32) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
      return 0x00002000;
    } else if (($array_size(VSXLEN) == 1 && VSXLEN[0] == 64) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN64)) {
      return 0x00003000;
    }
  }
}
```

D.166. 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.167. 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.168. 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.169. 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 && (CSR[mstatus].MXR == 1);
Boolean pbmte = implemented?(ExtensionName::Svpbmt) && CSR[menvcfg].PBMTE == 1;
Boolean adue = implemented?(ExtensionName::Svadu) && CSR[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 = CSR[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.170. 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 (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  adue = implemented?(ExtensionName::Svadu) && CSR[henvcfg].ADUE == 1;
} else {
  adue = implemented?(ExtensionName::Svadu) && CSR[menvcfg].ADUE == 1;
}
Boolean pbmte;
if (VA_SIZE == 32) {
  pbmte = false;
} else {
  if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    pbmte = implemented?(ExtensionName::Svpbmt) && CSR[henvcfg].PBMTE == 1;
  } else {
    pbmte = implemented?(ExtensionName::Svpbmt) && CSR[menvcfg].PBMTE == 1;
  }
```

```
}
Boolean mxr;
if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  mxr = (CSR[mstatus].MXR == 1) || (CSR[vsstatus].MXR == 1);
  ppn = CSR[vsatp].PPN;
} else {
  mxr = CSR[mstatus].MXR == 1;
  ppn = CSR[satp].PPN;
}
Boolean sum;
if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS)) {
  sum = CSR[vsstatus].SUM == 1;
} else {
  sum = CSR[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) {
    Bits<PA_SIZE> paddr_base = pte[PA_SIZE - 3:I * VPN_SIZE + 10] `<< (I * VPN_SIZE + 12);
    Bits<PA_SIZE> offset = vaddr[I * VPN_SIZE + 11:0];
    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 (CSR[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(paddr_base + offset, 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(paddr_base + offset, 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];
  }
}
```

D.171. 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 (xlen() == 32 && translation_mode == SatpMode::Sv32) {
  result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode, encoding);
} else if (xlen() == 64 && translation_mode == SatpMode::Sv39) {
  result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode, encoding);
} else if (xlen() == 64 && translation_mode == SatpMode::Sv48) {
  result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode, encoding);
} else if (xlen() == 64 && 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.172. 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 (CSR[misa].S == 1'b0) {
  return true;
}
SatpMode satp_mode;
if (virtual_mode?()) {
  satp_mode = $enum(SatpMode, CSR[vsatp].MODE);
} else {
  satp_mode = $enum(SatpMode, CSR[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.173. 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, CSR[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.174. 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.175. 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 (CSR[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.176. 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 = (CSR[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 = (CSR[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 == "sequential_bytes") {
    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.177. 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.178. 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.179. 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 (CSR[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.180. 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.181. 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.182. 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 = (CSR[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 (CSR[misa].S == 1 && LRSC_FAIL_ON_VA_SYNONYM) {
  reservation_virtual_address = virtual_address;
}
return read_memory_aligned<N>(physical_address, encoding);
```

D.183. 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 = (CSR[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.184. 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 = (CSR[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.185. 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 = (CSR[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.186. 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 = (CSR[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 = (CSR[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 == "sequential_bytes") {
    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.187. 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 = (CSR[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.188. 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.189. 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.190. 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) && CSR[mstateen0].JVT == 1'b0) || (mode() ==
PrivilegeMode::U && implemented?(ExtensionName::Ssstateen) && CSR[sstateen0].JVT == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else if ((mode() == PrivilegeMode::VS && implemented?(ExtensionName::Ssstateen) && CSR[hstateen0].JVT == 1'b0) || (mode() ==
PrivilegeMode::VU && implemented?(ExtensionName::Ssstateen) && (CSR[sstateen0].JVT == 1'b0 || CSR[hstateen0].JVT == 1'b0))) {
  raise(ExceptionCode::VirtualInstruction, mode(), encoding);
}
```