MC300-64 Processor Certification Requirements Document

Table of Contents

CRD Revision History 1
Typographic Conventions 1
Glossary 1
1. Introduction 2
2. Extensions 5
3. Implementation-dependencies 6
4. Traps 7
5. Instruction Summary 8
6. CSR Summary 11
Appendix A: Extension Details 16
Appendix B: Instruction Details 35
Appendix C: CSR Details 222
Appendix D: IDL Function Details 618

CRD Revision History

History of documentation changes that eventually lead to releases.

Date	Revision	Changes
2024-11-27	0.1.0	• First created

Typographic Conventions

CSR field colors

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

CSR field types

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

Glossary

Table 1. Glossary

Term	Meaning
RISC-V	(Reduced Instruction Set Computer) architecture, version 5
RVI	RISC-V International (organization that oversees RISC-V)
RVCP	RISC-V Certification Program
TSC	Technical Steering Committee (branch of RVI that creates standards)

Term	Meaning
CSC	Certification Steering Committee (branch of RVI that oversees the RVCP)
CRD	Certification Requirements Document
CTP	Certification Test Plan
CSR	Control & Status Register (located inside processor, not memory-mapped)
TBD	To Be Determined
N/A	“Not Applicable”
AKA	“Also Known As”
Must	Indicates a mandatory requirement
Should	Indicates a recommended requirement
May	Indicates an optional requirement

1. Introduction

The MC300 Processor Certificate targets advanced RISC-V microcontrollers. It supports either a 32-bit (MC300-32) or 64-bit (MC300-64) base ISA. The MC300 adds the following mandatory extensions to the MC200:

- S extension (Supervisor-mode privilege level)
- Sspmp extension (S-mode PMP, not ratified yet)

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
MC300-64	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 MC300-64 certificate doesn't cover their behaviors.

2.1. Mandatory Extensions

The MC300-64 certificate has 11 mandatory extensions.

Requirement ID	Extension	Version	Long Name	Note
REQ-EXT-B	B	~> 1.0.0	Bitmanipulation instructions	
REQ-EXT-C	C	~> 2.2	Compressed instructions	
REQ-EXT-I	I	~> 2.1	Base integer ISA (RV32I or RV64I)	
REQ-EXT-M	M	~> 2.0	Integer multiply and divide instructions	
REQ-EXT-S	S	~> 1.11.0	Supervisor mode	
REQ-EXT-Sm	Sm	~> 1.11.0	Machine mode	
REQ-EXT-Smpmp	Smpmp	~> 1.11.0	Physical Memory Protection	
REQ-EXT-U	U	~> 1.0.0	User-mode privilege level	
REQ-EXT-Zce	Zce	~> 1.0.0	Compressed instructions for microcontrollers	
REQ-EXT-Zicntr	Zicntr	~> 2.0	Architectural performance counters	
REQ-EXT-Zicsr	Zicsr	~> 2.0	Control and status registers	

2.2. Optional Extensions

None

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 MC300-64 certificate has 20 IN-SCOPE parameters.

Parameter	Type	Allowed Value(s)	Extension(s)	Note
ARCH_ID	64-bit integer	Any	Sm	
IMP_ID	64-bit integer	Any	Sm	
MISALIGNED_LDST	boolean	Any	Sm	
MISALIGNED_LDST_EXCEPTION_PRIORITY	[low, high]	Any	Sm	
MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE	[0, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]	Any	Sm	
MISALIGNED_SPLIT_STRATEGY	[by_byte, custom]	by_byte	Sm	
MISA_CSR_IMPLEMENTED	boolean	Any	Sm	
MTVAL_WIDTH	≤ 64	Any	Sm	
MTVEC_BASE_ALIGNMENT_DIRECT	4 to 64	Any	Sm	
MTVEC_BASE_ALIGNMENT_VECTORED	≥ 4	Any	Sm	
MTVEC_MODES	1-element to 2-element array of [0, 1]	Any	Sm	
MXLEN	[32, 64]	64	Sm	
M_MODE_ENDIANNESS	[little, big, dynamic]	little	Sm	
PHYS_ADDR_WIDTH	1 to 64	Any	Sm	
PRECISE_SYNCHRONOUS_EXCEPTIONS	boolean	true	Sm	
TIME_CSR_IMPLEMENTED	boolean	Any	Zicntr	
TRAP_ON_EBREAK	boolean	true	Sm	
TRAP_ON_ECALL_FROM_M	boolean	true	Sm	
VENDOR_ID_BANK	25-bit integer	Any	Sm	
VENDOR_ID_OFFSET	7-bit integer	Any	Sm	

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 MC300-64 certificate has 55 OUT-OF-SCOPE parameters.

Parameters	Type	Extension(s)
ASID_WIDTH	0 to 16	S
CONFIG_PTR_ADDRESS	64-bit integer	Sm
HSTATEEN_ENVCFG_TYPE	[rw, read-only-0, read-only-1]	S
MSTATEEN_ENVCFG_TYPE	[rw, read-only-0, read-only-1]	S
MSTATUS_FS_LEGAL_VALUES	at most 4-element array of [0, 1, 2, 3]	S
MSTATUS_FS_WRITABLE	boolean	S
MSTATUS_TVM_IMPLEMENTED	boolean	S
MSTATUS_VS_LEGAL_VALUES	at most 4-element array of [0, 1, 2, 3]	S
MSTATUS_VS_WRITABLE	boolean	S
MUTABLE_MISA_B	boolean	B
MUTABLE_MISA_C	boolean	C
MUTABLE_MISA_M	boolean	M
MUTABLE_MISA_S	boolean	S
MUTABLE_MISA_U	boolean	U

Parameters	Type	Extension(s)
NUM_PMP_ENTRIES	0 to 64	Smpmp
PMA_GRANULARITY	2 to 66	Sm
PMP_GRANULARITY	2 to 66	Smpmp
REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION	boolean	Sm
REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION	boolean	S
REPORT_VA_IN_MTVAL_ON_BREAKPOINT	boolean	Sm
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT	boolean	Sm
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED	boolean	Sm
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT	boolean	S
REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT	boolean	Sm
REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED	boolean	Sm
REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT	boolean	S
REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT	boolean	Sm
REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED	boolean	Sm
REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT	boolean	S
REPORT_VA_IN_STVAL_ON_BREAKPOINT	boolean	S
REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT	boolean	S
REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED	boolean	S
REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT	boolean	S
REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT	boolean	S
REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED	boolean	S
REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT	boolean	S
REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT	boolean	S
REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED	boolean	S
REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT	boolean	S
SATP_MODE_BARE	boolean	S
SCOUNTENABLE_EN	32-element array of boolean	S
STVAL_WIDTH	≤ 0xffffffffffffffff	S
STVEC_MODE_DIRECT	boolean	S
STVEC_MODE_VECTORED	boolean	S
SXLEN	[32, 64, 3264]	S
S_MODE_ENDIANNESS	[little, big, dynamic]	S
TRAP_ON_ECALL_FROM_S	boolean	S
TRAP_ON_ECALL_FROM_U	boolean	U
TRAP_ON_ILLEGAL_WLRL	boolean	Sm
TRAP_ON_RESERVED_INSTRUCTION	boolean	Sm
TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY	boolean	S
TRAP_ON_UNIMPLEMENTED_CSR	boolean	Sm
TRAP_ON_UNIMPLEMENTED_INSTRUCTION	boolean	Sm
UXLEN	[32, 64, 3264]	U
U_MODE_ENDIANNESS	[little, big, dynamic]	U

4. Traps

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

4.1. Synchronous Exceptions

`xcause.CODE` CSR Field Value	Name
0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction

3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store/AMO address misaligned
7	Store/AMO access fault
8	Environment call from <%- if ext?(:H) -%>V<%- end -%>U-mode
9	Environment call from <%- if ext?(:H) -%>H<%- end -%>S-mode
11	Environment call from M-mode
12	Instruction page fault
13	Load page fault
15	Store/AMO page fault

4.2. Asynchronous Interrupts

`xcause.CODE` CSR Field Value	Name
1	Supervisor software interrupt
2	Virtual supervisor software interrupt
3	Machine software interrupt
5	Supervisor timer interrupt
6	Virtual supervisor timer interrupt
7	Machine timer interrupt
9	Supervisor external interrupt
10	Virtual supervisor external interrupt
11	Machine external interrupt
12	Supervisor guest external interrupt
13	Local counter overflow interrupt

5. Instruction Summary

The MC300-64 certificate has up to 123 instructions (exact number depends on an implementation's options).

Name	Long Name
add	Integer add
add.uw	Add unsigned word
addi	Add immediate
addiw	Add immediate word
addw	Add word
and	And
andi	And immediate
andn	AND with inverted operand
auipc	Add upper immediate to pc
bclr	Single-Bit clear (Register)
bclri	Single-Bit clear (Immediate)
beq	Branch if equal
bext	Single-Bit extract (Register)
bexti	Single-Bit extract (Immediate)
bge	Branch if greater than or equal
bgeu	Branch if greater than or equal unsigned
binv	Single-Bit invert (Register)
binvi	Single-Bit invert (Immediate)
blt	Branch if less than
bltu	Branch if less than unsigned
bne	Branch if not equal
bset	Single-Bit set (Register)

Name	Long Name
bseti	Single-Bit set (Immediate)
c.fld	Load double-precision
c.fldsp	Load doubleword into floating-point register from stack
c.flw	Load single-precision
c.flwsp	Load word into floating-point register from stack
c.fsd	Store double-precision
c.fsdsp	Store double-precision value to stack
c.fsw	Store single-precision
c.fswsp	Store single-precision value to stack
clz	Count leading zero bits
clzw	Count leading zero bits in word
cpop	Count set bits
cpopw	Count set bits in word
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
ctz	Count trailing zero bits
ctzw	Count trailing zero bits in word
div	Signed division
divu	Unsigned division
divuw	Unsigned 32-bit division
divw	Signed 32-bit division
ebreak	Breakpoint exception
ecall	Environment call
fence	Memory oxdering 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
lwu	Load word unsigned
max	Maximum
maxu	Unsigned maximum
min	Minimum
minu	Unsigned minimum
mret	Machine Exception Return
mul	Signed multiply
mulh	Signed multiply high
mulhsu	Signed/unsigned multiply high
mulhu	Unsigned multiply high
mulw	Signed 32-bit multiply
or	Or
orc.b	Bitware OR-combine, byte granule
ori	Or immediate

Name	Long Name
orn	OR with inverted operand
rem	Signed remainder
remu	Unsigned remainder
remuw	Unsigned 32-bit remainder
remw	Signed 32-bit remainder
rev8	Byte-reverse register (RV64 encoding)
rol	Rotate left (Register)
rolw	Rotate left word (Register)
ror	Rotate right (Register)
rori	Rotate right (Immediate)
roriw	Rotate right word (Immediate)
rorw	Rotate right word (Register)
sb	Store byte
sd	Store doubleword
sext.b	Sign-extend byte
sext.h	Sign-extend halfword
sfence.vma	Supervisor memory-management fence
sh	Store halfword
sh1add	Shift left by 1 and add
sh1add.uw	Shift unsigned word left by 1 and add
sh2add	Shift left by 2 and add
sh2add.uw	Shift unsigned word left by 2 and add
sh3add	Shift left by 3 and add
sh3add.uw	Shift unsigned word left by 3 and add
sll	Shift left logical
slli	Shift left logical immediate
slli.uw	Shift left unsigned word (Immediate)
slliw	Shift left logical immediate word
sllw	Shift left logical word
slt	Set on less than
slti	Set on less than immediate
sltiu	Set on less than immediate unsigned
sltu	Set on less than unsigned
sra	Shift right arithmetic
srai	Shift right arithmetic immediate
sraiw	Shift right arithmetic immediate word
sraw	Shift right arithmetic word
sret	Supervisor Exception Return
srl	Shift right logical
srli	Shift right logical immediate
srliw	Shift right logical immediate word
srlw	Shift right logical word
sub	Subtract
subw	Subtract word
sw	Store word
wfi	Wait for interrupt
xnor	Exclusive NOR
xor	Exclusive Or
xori	Exclusive Or immediate

6. CSR Summary

The MC300-64 certificate has up to 120 CSRs (exact number depends on an implementation's options).

6.1. By Name

Name	Long Name	Address	Mode	Primary Extension
cycle	Cycle counter for RDCYCLE Instruction	0xc00	U	Zicntr ~> 2.0.0
cycleh	High-half cycle counter for RDCYCLE Instruction	0xc80	U	Zicntr ~> 2.0.0
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U	Zicntr ~> 2.0.0
instreth	Instructions retired counter, high bits	0xc82	U	Zicntr ~> 2.0.0
marchid	Machine Architecture ID	0xf12	M	Sm ~> 1.11.0
mcause	Machine Cause	0x342	M	Sm ~> 1.11.0
mcounteren	Machine Counter Enable	0x306	M	U ~> 1.0.0
mcountinhibit	Machine Counter Inhibit	0x320	M	Sm ~> 1.11.0
mcycle	Machine Cycle Counter	0xb00	M	Zicntr ~> 2.0.0
mcycleh	High-half machine Cycle Counter	0xb80	M	Zicntr ~> 2.0.0
medeleg	Machine Exception Delegation	0x302	M	S ~> 1.11.0
menvcfg	Machine Environment Configuration	0x30a	M	Sm >= 1.12
menvcfgh	Machine Environment Configuration	0x31a	M	Sm >= 1.12
mepc	Machine Exception Program Counter	0x341	M	Sm ~> 1.11.0
mhartid	Machine Hart ID	0xf14	M	Sm ~> 1.11.0
mideleg	Machine Interrupt Delegation	0x303	M	Sm ⇐ 1.9.1
mie	Machine Interrupt Enable	0x304	M	Sm ~> 1.11.0
mimpid	Machine Implementation ID	0xf13	M	Sm ~> 1.11.0
minstret	Machine Instructions Retired Counter	0xb02	M	Zicntr ~> 2.0.0
minstreth	Machine Instructions Retired Counter	0xb82	M	Zicntr ~> 2.0.0
mip	Machine Interrupt Pending	0x344	M	Sm ~> 1.11.0
misa	Machine ISA Control	0x301	M	Sm ~> 1.11.0
mnepc	Machine Exception Program Counter	0x741	M	Sm ~> 1.11.0
mscratch	Machine Scratch Register	0x340	M	Sm ~> 1.11.0
mstatus	Machine Status	0x300	M	Sm ~> 1.11.0
mtval	Machine Trap Value	0x343	M	Sm ~> 1.11.0
mtvec	Machine Trap Vector Control	0x305	M	Sm ~> 1.11.0
mvendorid	Machine Vendor ID	0xf11	M	Sm ~> 1.11.0
pmpaddr0	PMP Address 0	0x3b0	M	Smpmp ~> 1.11.0
pmpaddr1	PMP Address 1	0x3b1	M	Smpmp ~> 1.11.0
pmpaddr10	PMP Address 10	0x3ba	M	Smpmp ~> 1.11.0
pmpaddr11	PMP Address 11	0x3bb	M	Smpmp ~> 1.11.0
pmpaddr12	PMP Address 12	0x3bc	M	Smpmp ~> 1.11.0
pmpaddr13	PMP Address 13	0x3bd	M	Smpmp ~> 1.11.0
pmpaddr14	PMP Address 14	0x3be	M	Smpmp ~> 1.11.0
pmpaddr15	PMP Address 15	0x3bf	M	Smpmp ~> 1.11.0
pmpaddr16	PMP Address 16	0x3c0	M	Smpmp ~> 1.11.0
pmpaddr17	PMP Address 17	0x3c1	M	Smpmp ~> 1.11.0
pmpaddr18	PMP Address 18	0x3c2	M	Smpmp ~> 1.11.0
pmpaddr19	PMP Address 19	0x3c3	M	Smpmp ~> 1.11.0
pmpaddr2	PMP Address 2	0x3b2	M	Smpmp ~> 1.11.0
pmpaddr20	PMP Address 20	0x3c4	M	Smpmp ~> 1.11.0
pmpaddr21	PMP Address 21	0x3c5	M	Smpmp ~> 1.11.0
pmpaddr22	PMP Address 22	0x3c6	M	Smpmp ~> 1.11.0
pmpaddr23	PMP Address 23	0x3c7	M	Smpmp ~> 1.11.0
pmpaddr24	PMP Address 24	0x3c8	M	Smpmp ~> 1.11.0
pmpaddr25	PMP Address 25	0x3c9	M	Smpmp ~> 1.11.0

Name	Long Name	Address	Mode	Primary Extension
pmpaddr26	PMP Address 26	0x3ca	M	Smpmp ~> 1.11.0
pmpaddr27	PMP Address 27	0x3cb	M	Smpmp ~> 1.11.0
pmpaddr28	PMP Address 28	0x3cc	M	Smpmp ~> 1.11.0
pmpaddr29	PMP Address 29	0x3cd	M	Smpmp ~> 1.11.0
pmpaddr3	PMP Address 3	0x3b3	M	Smpmp ~> 1.11.0
pmpaddr30	PMP Address 30	0x3ce	M	Smpmp ~> 1.11.0
pmpaddr31	PMP Address 31	0x3cf	M	Smpmp ~> 1.11.0
pmpaddr32	PMP Address 32	0x3d0	M	Smpmp ~> 1.11.0
pmpaddr33	PMP Address 33	0x3d1	M	Smpmp ~> 1.11.0
pmpaddr34	PMP Address 34	0x3d2	M	Smpmp ~> 1.11.0
pmpaddr35	PMP Address 35	0x3d3	M	Smpmp ~> 1.11.0
pmpaddr36	PMP Address 36	0x3d4	M	Smpmp ~> 1.11.0
pmpaddr37	PMP Address 37	0x3d5	M	Smpmp ~> 1.11.0
pmpaddr38	PMP Address 38	0x3d6	M	Smpmp ~> 1.11.0
pmpaddr39	PMP Address 39	0x3d7	M	Smpmp ~> 1.11.0
pmpaddr4	PMP Address 4	0x3b4	M	Smpmp ~> 1.11.0
pmpaddr40	PMP Address 40	0x3d8	M	Smpmp ~> 1.11.0
pmpaddr41	PMP Address 41	0x3d9	M	Smpmp ~> 1.11.0
pmpaddr42	PMP Address 42	0x3da	M	Smpmp ~> 1.11.0
pmpaddr43	PMP Address 43	0x3db	M	Smpmp ~> 1.11.0
pmpaddr44	PMP Address 44	0x3dc	M	Smpmp ~> 1.11.0
pmpaddr45	PMP Address 45	0x3dd	M	Smpmp ~> 1.11.0
pmpaddr46	PMP Address 46	0x3de	M	Smpmp ~> 1.11.0
pmpaddr47	PMP Address 47	0x3df	M	Smpmp ~> 1.11.0
pmpaddr48	PMP Address 48	0x3e0	M	Smpmp ~> 1.11.0
pmpaddr49	PMP Address 49	0x3e1	M	Smpmp ~> 1.11.0
pmpaddr5	PMP Address 5	0x3b5	M	Smpmp ~> 1.11.0
pmpaddr50	PMP Address 50	0x3e2	M	Smpmp ~> 1.11.0
pmpaddr51	PMP Address 51	0x3e3	M	Smpmp ~> 1.11.0
pmpaddr52	PMP Address 52	0x3e4	M	Smpmp ~> 1.11.0
pmpaddr53	PMP Address 53	0x3e5	M	Smpmp ~> 1.11.0
pmpaddr54	PMP Address 54	0x3e6	M	Smpmp ~> 1.11.0
pmpaddr55	PMP Address 55	0x3e7	M	Smpmp ~> 1.11.0
pmpaddr56	PMP Address 56	0x3e8	M	Smpmp ~> 1.11.0
pmpaddr57	PMP Address 57	0x3e9	M	Smpmp ~> 1.11.0
pmpaddr58	PMP Address 58	0x3ea	M	Smpmp ~> 1.11.0
pmpaddr59	PMP Address 59	0x3eb	M	Smpmp ~> 1.11.0
pmpaddr6	PMP Address 6	0x3b6	M	Smpmp ~> 1.11.0
pmpaddr60	PMP Address 60	0x3ec	M	Smpmp ~> 1.11.0
pmpaddr61	PMP Address 61	0x3ed	M	Smpmp ~> 1.11.0
pmpaddr62	PMP Address 62	0x3ee	M	Smpmp ~> 1.11.0
pmpaddr63	PMP Address 63	0x3ef	M	Smpmp ~> 1.11.0
pmpaddr7	PMP Address 7	0x3b7	M	Smpmp ~> 1.11.0
pmpaddr8	PMP Address 8	0x3b8	M	Smpmp ~> 1.11.0
pmpaddr9	PMP Address 9	0x3b9	M	Smpmp ~> 1.11.0
pmpcfg0	PMP Configuration Register 0	0x3a0	M	Smpmp ~> 1.11.0
pmpcfg1	PMP Configuration Register 1	0x3a1	M	Smpmp ~> 1.11.0
pmpcfg10	PMP Configuration Register 10	0x3aa	M	Smpmp ~> 1.11.0
pmpcfg11	PMP Configuration Register 11	0x3ab	M	Smpmp ~> 1.11.0
pmpcfg12	PMP Configuration Register 12	0x3ac	M	Smpmp ~> 1.11.0
pmpcfg13	PMP Configuration Register 13	0x3ad	M	Smpmp ~> 1.11.0
pmpcfg14	PMP Configuration Register 14	0x3ae	M	Smpmp ~> 1.11.0

Name	Long Name	Address	Mode	Primary Extension
pmpcfg15	PMP Configuration Register 15	0x3af	M	Smpmp ~> 1.11.0
pmpcfg2	PMP Configuration Register 2	0x3a2	M	Smpmp ~> 1.11.0
pmpcfg3	PMP Configuration Register 3	0x3a3	M	Smpmp ~> 1.11.0
pmpcfg4	PMP Configuration Register 4	0x3a4	M	Smpmp ~> 1.11.0
pmpcfg5	PMP Configuration Register 5	0x3a5	M	Smpmp ~> 1.11.0
pmpcfg6	PMP Configuration Register 6	0x3a6	M	Smpmp ~> 1.11.0
pmpcfg7	PMP Configuration Register 7	0x3a7	M	Smpmp ~> 1.11.0
pmpcfg8	PMP Configuration Register 8	0x3a8	M	Smpmp ~> 1.11.0
pmpcfg9	PMP Configuration Register 9	0x3a9	M	Smpmp ~> 1.11.0
satp	Supervisor Address Translation and Protection	0x180	S	S ~> 1.11.0
scause	Supervisor Cause	0x142	S	S ~> 1.11.0
scounteren	Supervisor Counter Enable	0x106	S	S ~> 1.11.0
senvcfg	Supervisor Environment Configuration	0x10a	S	S >= 1.12
sepc	Supervisor Exception Program Counter	0x141	S	S ~> 1.11.0
sip	Supervisor Interrupt Pending	0x144	S	S ~> 1.11.0
sscratch	Supervisor Scratch Register	0x140	S	S ~> 1.11.0
sstatus	Supervisor Status	0x100	S	S ~> 1.11.0
stval	Supervisor Trap Value	0x143	S	S ~> 1.11.0
stvec	Supervisor Trap Vector	0x105	S	S ~> 1.11.0
time	Timer for RDTIME Instruction	0xc01	U	Zicntr ~> 2.0.0
timeh	High-half timer for RDTIME Instruction	0xc81	U	Zicntr ~> 2.0.0

6.2. By Address

Address	Mode	Name	Long Name	Primary Extension
0x100	S	sstatus	Supervisor Status	S ~> 1.11.0
0x105	S	stvec	Supervisor Trap Vector	S ~> 1.11.0
0x106	S	scounteren	Supervisor Counter Enable	S ~> 1.11.0
0x10a	S	senvcfg	Supervisor Environment Configuration	S >= 1.12
0x140	S	sscratch	Supervisor Scratch Register	S ~> 1.11.0
0x141	S	sepc	Supervisor Exception Program Counter	S ~> 1.11.0
0x142	S	scause	Supervisor Cause	S ~> 1.11.0
0x143	S	stval	Supervisor Trap Value	S ~> 1.11.0
0x144	S	sip	Supervisor Interrupt Pending	S ~> 1.11.0
0x180	S	satp	Supervisor Address Translation and Protection	S ~> 1.11.0
0x300	M	mstatus	Machine Status	Sm ~> 1.11.0
0x301	M	misa	Machine ISA Control	Sm ~> 1.11.0
0x302	M	medeleg	Machine Exception Delegation	S ~> 1.11.0
0x303	M	mideleg	Machine Interrupt Delegation	Sm ⇐ 1.9.1
0x304	M	mie	Machine Interrupt Enable	Sm ~> 1.11.0
0x305	M	mtvec	Machine Trap Vector Control	Sm ~> 1.11.0
0x306	M	mcounteren	Machine Counter Enable	U ~> 1.0.0
0x30a	M	menvcfg	Machine Environment Configuration	Sm >= 1.12
0x31a	M	menvcfgh	Machine Environment Configuration	Sm >= 1.12
0x320	M	mcountinhibit	Machine Counter Inhibit	Sm ~> 1.11.0
0x340	M	mscratch	Machine Scratch Register	Sm ~> 1.11.0
0x341	M	mepc	Machine Exception Program Counter	Sm ~> 1.11.0
0x342	M	mcause	Machine Cause	Sm ~> 1.11.0
0x343	M	mtval	Machine Trap Value	Sm ~> 1.11.0
0x344	M	mip	Machine Interrupt Pending	Sm ~> 1.11.0
0x3a0	M	pmpcfg0	PMP Configuration Register 0	Smpmp ~> 1.11.0
0x3a1	M	pmpcfg1	PMP Configuration Register 1	Smpmp ~> 1.11.0

Address	Mode	Name	Long Name	Primary Extension
0x3a2	M	pmpcfg2	PMP Configuration Register 2	Smpmp ~> 1.11.0
0x3a3	M	pmpcfg3	PMP Configuration Register 3	Smpmp ~> 1.11.0
0x3a4	M	pmpcfg4	PMP Configuration Register 4	Smpmp ~> 1.11.0
0x3a5	M	pmpcfg5	PMP Configuration Register 5	Smpmp ~> 1.11.0
0x3a6	M	pmpcfg6	PMP Configuration Register 6	Smpmp ~> 1.11.0
0x3a7	M	pmpcfg7	PMP Configuration Register 7	Smpmp ~> 1.11.0
0x3a8	M	pmpcfg8	PMP Configuration Register 8	Smpmp ~> 1.11.0
0x3a9	M	pmpcfg9	PMP Configuration Register 9	Smpmp ~> 1.11.0
0x3aa	M	pmpcfg10	PMP Configuration Register 10	Smpmp ~> 1.11.0
0x3ab	M	pmpcfg11	PMP Configuration Register 11	Smpmp ~> 1.11.0
0x3ac	M	pmpcfg12	PMP Configuration Register 12	Smpmp ~> 1.11.0
0x3ad	M	pmpcfg13	PMP Configuration Register 13	Smpmp ~> 1.11.0
0x3ae	M	pmpcfg14	PMP Configuration Register 14	Smpmp ~> 1.11.0
0x3af	M	pmpcfg15	PMP Configuration Register 15	Smpmp ~> 1.11.0
0x3b0	M	pmpaddr0	PMP Address 0	Smpmp ~> 1.11.0
0x3b1	M	pmpaddr1	PMP Address 1	Smpmp ~> 1.11.0
0x3b2	M	pmpaddr2	PMP Address 2	Smpmp ~> 1.11.0
0x3b3	M	pmpaddr3	PMP Address 3	Smpmp ~> 1.11.0
0x3b4	M	pmpaddr4	PMP Address 4	Smpmp ~> 1.11.0
0x3b5	M	pmpaddr5	PMP Address 5	Smpmp ~> 1.11.0
0x3b6	M	pmpaddr6	PMP Address 6	Smpmp ~> 1.11.0
0x3b7	M	pmpaddr7	PMP Address 7	Smpmp ~> 1.11.0
0x3b8	M	pmpaddr8	PMP Address 8	Smpmp ~> 1.11.0
0x3b9	M	pmpaddr9	PMP Address 9	Smpmp ~> 1.11.0
0x3ba	M	pmpaddr10	PMP Address 10	Smpmp ~> 1.11.0
0x3bb	M	pmpaddr11	PMP Address 11	Smpmp ~> 1.11.0
0x3bc	M	pmpaddr12	PMP Address 12	Smpmp ~> 1.11.0
0x3bd	M	pmpaddr13	PMP Address 13	Smpmp ~> 1.11.0
0x3be	M	pmpaddr14	PMP Address 14	Smpmp ~> 1.11.0
0x3bf	M	pmpaddr15	PMP Address 15	Smpmp ~> 1.11.0
0x3c0	M	pmpaddr16	PMP Address 16	Smpmp ~> 1.11.0
0x3c1	M	pmpaddr17	PMP Address 17	Smpmp ~> 1.11.0
0x3c2	M	pmpaddr18	PMP Address 18	Smpmp ~> 1.11.0
0x3c3	M	pmpaddr19	PMP Address 19	Smpmp ~> 1.11.0
0x3c4	M	pmpaddr20	PMP Address 20	Smpmp ~> 1.11.0
0x3c5	M	pmpaddr21	PMP Address 21	Smpmp ~> 1.11.0
0x3c6	M	pmpaddr22	PMP Address 22	Smpmp ~> 1.11.0
0x3c7	M	pmpaddr23	PMP Address 23	Smpmp ~> 1.11.0
0x3c8	M	pmpaddr24	PMP Address 24	Smpmp ~> 1.11.0
0x3c9	M	pmpaddr25	PMP Address 25	Smpmp ~> 1.11.0
0x3ca	M	pmpaddr26	PMP Address 26	Smpmp ~> 1.11.0
0x3cb	M	pmpaddr27	PMP Address 27	Smpmp ~> 1.11.0
0x3cc	M	pmpaddr28	PMP Address 28	Smpmp ~> 1.11.0
0x3cd	M	pmpaddr29	PMP Address 29	Smpmp ~> 1.11.0
0x3ce	M	pmpaddr30	PMP Address 30	Smpmp ~> 1.11.0
0x3cf	M	pmpaddr31	PMP Address 31	Smpmp ~> 1.11.0
0x3d0	M	pmpaddr32	PMP Address 32	Smpmp ~> 1.11.0
0x3d1	M	pmpaddr33	PMP Address 33	Smpmp ~> 1.11.0
0x3d2	M	pmpaddr34	PMP Address 34	Smpmp ~> 1.11.0
0x3d3	M	pmpaddr35	PMP Address 35	Smpmp ~> 1.11.0
0x3d4	M	pmpaddr36	PMP Address 36	Smpmp ~> 1.11.0
0x3d5	M	pmpaddr37	PMP Address 37	Smpmp ~> 1.11.0

Address	Mode	Name	Long Name	Primary Extension
0x3d6	M	pmpaddr38	PMP Address 38	Smpmp ~> 1.11.0
0x3d7	M	pmpaddr39	PMP Address 39	Smpmp ~> 1.11.0
0x3d8	M	pmpaddr40	PMP Address 40	Smpmp ~> 1.11.0
0x3d9	M	pmpaddr41	PMP Address 41	Smpmp ~> 1.11.0
0x3da	M	pmpaddr42	PMP Address 42	Smpmp ~> 1.11.0
0x3db	M	pmpaddr43	PMP Address 43	Smpmp ~> 1.11.0
0x3dc	M	pmpaddr44	PMP Address 44	Smpmp ~> 1.11.0
0x3dd	M	pmpaddr45	PMP Address 45	Smpmp ~> 1.11.0
0x3de	M	pmpaddr46	PMP Address 46	Smpmp ~> 1.11.0
0x3df	M	pmpaddr47	PMP Address 47	Smpmp ~> 1.11.0
0x3e0	M	pmpaddr48	PMP Address 48	Smpmp ~> 1.11.0
0x3e1	M	pmpaddr49	PMP Address 49	Smpmp ~> 1.11.0
0x3e2	M	pmpaddr50	PMP Address 50	Smpmp ~> 1.11.0
0x3e3	M	pmpaddr51	PMP Address 51	Smpmp ~> 1.11.0
0x3e4	M	pmpaddr52	PMP Address 52	Smpmp ~> 1.11.0
0x3e5	M	pmpaddr53	PMP Address 53	Smpmp ~> 1.11.0
0x3e6	M	pmpaddr54	PMP Address 54	Smpmp ~> 1.11.0
0x3e7	M	pmpaddr55	PMP Address 55	Smpmp ~> 1.11.0
0x3e8	M	pmpaddr56	PMP Address 56	Smpmp ~> 1.11.0
0x3e9	M	pmpaddr57	PMP Address 57	Smpmp ~> 1.11.0
0x3ea	M	pmpaddr58	PMP Address 58	Smpmp ~> 1.11.0
0x3eb	M	pmpaddr59	PMP Address 59	Smpmp ~> 1.11.0
0x3ec	M	pmpaddr60	PMP Address 60	Smpmp ~> 1.11.0
0x3ed	M	pmpaddr61	PMP Address 61	Smpmp ~> 1.11.0
0x3ee	M	pmpaddr62	PMP Address 62	Smpmp ~> 1.11.0
0x3ef	M	pmpaddr63	PMP Address 63	Smpmp ~> 1.11.0
0x741	M	mnepc	Machine Exception Program Counter	Sm ~> 1.11.0
0xb00	M	mcycle	Machine Cycle Counter	Zicntr ~> 2.0.0
0xb02	M	minstret	Machine Instructions Retired Counter	Zicntr ~> 2.0.0
0xb80	M	mcycleh	High-half machine Cycle Counter	Zicntr ~> 2.0.0
0xb82	M	minstreth	Machine Instructions Retired Counter	Zicntr ~> 2.0.0
0xc00	U	cycle	Cycle counter for RDCYCLE Instruction	Zicntr ~> 2.0.0
0xc01	U	time	Timer for RDTIME Instruction	Zicntr ~> 2.0.0
0xc02	U	instret	Instructions retired counter for RDINSTRET Instruction	Zicntr ~> 2.0.0
0xc80	U	cycleh	High-half cycle counter for RDCYCLE Instruction	Zicntr ~> 2.0.0
0xc81	U	timeh	High-half timer for RDTIME Instruction	Zicntr ~> 2.0.0
0xc82	U	instreth	Instructions retired counter, high bits	Zicntr ~> 2.0.0
0xf11	M	mvendorid	Machine Vendor ID	Sm ~> 1.11.0
0xf12	M	marchid	Machine Architecture ID	Sm ~> 1.11.0
0xf13	M	mimpid	Machine Implementation ID	Sm ~> 1.11.0
0xf14	M	mhartid	Machine Hart ID	Sm ~> 1.11.0

Appendix A: Extension Details

A.1. Extension B

Long Name: Bitmanipulation instructions
Version Requirement: ~> 1.0.0

1.0.0

State

ratified

Ratification date

2024-04

Ratification document

drive.google.com/file/d/1SgLoasaBjs5WboQMaU3wpHkjUwV71UZn/view

Implies

- Zba version 1.0.0
- Zbb version 1.0.0
- Zbs version 1.0.0

A.1.1. Synopsis

The B standard extension comprises instructions provided by the Zba, Zbb, and Zbs extensions.

Bit 1 of the misa register encodes the presence of the B standard extension. When misa.B is 1, the implementation supports the instructions provided by the Zba, Zbb, and Zbs extensions. When misa.B is 0, it indicates that the implementation may not support one or more of the Zba, Zbb, or Zbs extensions.

A.1.2. Instructions

The following 39 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

andn	**AND with inverted operand**
orn	**OR with inverted operand**
rev8	**Byte-reverse register (RV64 encoding)**
rol	**Rotate left (Register)**
rolw	**Rotate left word (Register)**
ror	**Rotate right (Register)**
rori	**Rotate right (Immediate)**
roriw	**Rotate right word (Immediate)**
rorw	**Rotate right word (Register)**
xnor	**Exclusive NOR**
add.uw	**Add unsigned word**
sh1add.uw	**Shift unsigned word left by 1 and add**
sh1add	**Shift left by 1 and add**
sh2add.uw	**Shift unsigned word left by 2 and add**
sh2add	**Shift left by 2 and add**
sh3add.uw	**Shift unsigned word left by 3 and add**
sh3add	**Shift left by 3 and add**
slli.uw	**Shift left unsigned word (Immediate)**
clz	**Count leading zero bits**
clzw	**Count leading zero bits in word**
cpop	**Count set bits**
cpopw	**Count set bits in word**
ctz	**Count trailing zero bits**
ctzw	**Count trailing zero bits in word**
max	**Maximum**
maxu	**Unsigned maximum**
min	**Minimum**
minu	**Unsigned minimum**

orc.b	**Bitware OR-combine, byte granule**
sext.b	**Sign-extend byte**
sext.h	**Sign-extend halfword**
bclr	**Single-Bit clear (Register)**
bclri	**Single-Bit clear (Immediate)**
bext	**Single-Bit extract (Register)**
bexti	**Single-Bit extract (Immediate)**
binv	**Single-Bit invert (Register)**
binvi	**Single-Bit invert (Immediate)**
bset	**Single-Bit set (Register)**
bseti	**Single-Bit set (Immediate)**

A.1.3. OUT-OF-SCOPE Parameters

MUTABLE_MISA_B ⇒ boolean

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

A.2. Extension C

Long Name: Compressed instructions
Version Requirement: ~> 2.2

2.0.0

State

ratified

Ratification date

2019-12

Implies

- Zca version 1.0.0
- Zcf version 1.0.0
- Zcd version 1.0.0

A.2.1. Synopsis

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

A.2.2. Overview

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

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

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

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

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

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

> *Although single-precision loads and stores are not a significant source of static or dynamic compression for benchmarks compiled for the currently supported ABIs, for microcontrollers that only provide hardware single-precision floating-point units and have an ABI that only supports single-precision floating-point numbers, the single-precision loads and stores will be used at least as frequently as double-precision loads and stores in the measured benchmarks. Hence, the motivation to provide compressed support for these in RV32C.*
>
> *Short-range subroutine calls are more likely in small binaries for microcontrollers, hence the motivation to include these in RV32C.*
>
> *Although reusing opcodes for different purposes for different base ISAs adds some complexity to documentation, the impact on implementation complexity is small even for designs that support multiple base ISAs. The compressed floating-point load and store variants use the same instruction format with the same register specifiers as the wider integer loads and stores.*

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

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

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

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

> *Variable-length instruction sets have long been used to improve code density. For example, the IBM Stretch cite:[stretch], developed in the late 1950s, had an ISA with 32-bit and 64-bit instructions, where some of the 32-bit instructions were compressed versions of the full 64-bit instructions. Stretch also employed the concept of limiting the set of registers that were addressable in some of the shorter instruction formats, with short branch instructions that could only refer to one of the index registers. The later IBM 360 architecture cite:[ibm360] supported a simple variable-length instruction encoding with 16-bit, 32-bit, or 48-bit instruction formats.*
>
> *In 1963, CDC introduced the Cray-designed CDC 6600 cite:[cdc6600], a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.*
>
> *The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced code size relative to their starting points by about 25-30%, yielding code that was significantly smaller than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.*
>
> *Since the original RISC ISAs did not leave sufficient opcode space free to include these unplanned compressed instructions, they were instead developed as complete new ISAs. This meant compilers needed different code generators for the separate compressed ISAs. The first compressed RISC ISA extensions (e.g., ARM Thumb and MIPS16) used only a fixed 16-bit instruction size, which gave good reductions in static code size but caused an increase in dynamic instruction count, which led to lower performance compared to the original fixed-width 32-bit instruction size. This led to the development of a second generation of compressed RISC ISA designs with mixed 16-bit and 32-bit instruction lengths (e.g., ARM Thumb2, microMIPS, PowerPC VLE), so that performance was similar to pure 32-bit instructions but with significant code size savings. Unfortunately, these different generations of compressed ISAs are incompatible with each other and with the original uncompressed ISA, leading to significant complexity in documentation, implementations, and software tools support.*
>
> *Of the commonly used 64-bit ISAs, only PowerPC and microMIPS currently supports a compressed instruction format. It is surprising that the most popular 64-bit ISA for mobile platforms (ARM v8) does not include a compressed instruction format given that static code size and dynamic instruction fetch bandwidth are important metrics. Although static code size is not a major concern in larger systems, instruction fetch bandwidth can be a major bottleneck in servers running commercial workloads, which often have a large instruction working set.*
>
> *Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications and to improve performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size. cite:[waterman-ms]*

A.2.3. Compressed Instruction Formats

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

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

which only have 16 integer registers.

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

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

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

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

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

Table 4. Compressed 16-bit RVC instruction formats

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

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

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

A.2.4. Instructions

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

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

A.2.5. OUT-OF-SCOPE Parameters

MUTABLE_MISA_C ⇒ boolean

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

A.3. Extension I

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

2.1.0

State

ratified

Ratification date

2019-06

Changes

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

A.3.1. Synopsis

Base integer instructions — TODO

A.3.2. Instructions

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

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

sraiw	**Shift right arithmetic immediate word**
sraw	**Shift right arithmetic word**
srl	**Shift right logical**
srli	**Shift right logical immediate**
srliw	**Shift right logical immediate word**
srlw	**Shift right logical word**
sub	**Subtract**
subw	**Subtract word**
sw	**Store word**
xor	**Exclusive Or**
xori	**Exclusive Or immediate**

A.4. Extension M

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

2.0.0

State
ratified

Ratification date
2019-12

A.4.1. Synopsis

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

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

A.4.2. Instructions

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

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

A.4.3. OUT-OF-SCOPE Parameters

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

A.5. Extension S

Long Name: Supervisor mode
Version Requirement: ~> 1.11.0

1.11.0

State

ratified

Ratification date

2019-06

1.12.0

State

ratified

Ratification date

2021-12

1.13.0

State

ratified

A.5.1. Synopsis

This chapter describes the RISC-V supervisor-level architecture, which contains a common core that is used with various supervisor-level address translation and protection schemes.

Supervisor mode is deliberately restricted in terms of interactions with underlying physical hardware, such as physical memory and device interrupts, to support clean virtualization. In this spirit, certain supervisor-level facilities, including requests for timer and interprocessor interrupts, are provided by implementation-specific mechanisms. In some systems, a supervisor execution environment (SEE) provides these facilities in a manner specified by a supervisor binary interface (SBI). Other systems supply these facilities directly, through some other implementation-defined mechanism.

A.5.2. Instructions

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

sfence.vma	**Supervisor memory-management fence**
sret	**Supervisor Exception Return**

A.5.3. CSRs

The following 11 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
medeleg	Machine Exception Delegation	0x302	M
mideleg	Machine Interrupt Delegation	0x303	M
satp	Supervisor Address Translation and Protection	0x180	S
scause	Supervisor Cause	0x142	S
scounteren	Supervisor Counter Enable	0x106	S
sepc	Supervisor Exception Program Counter	0x141	S
sip	Supervisor Interrupt Pending	0x144	S
sscratch	Supervisor Scratch Register	0x140	S
sstatus	Supervisor Status	0x100	S
stval	Supervisor Trap Value	0x143	S
stvec	Supervisor Trap Vector	0x105	S

A.5.4. OUT-OF-SCOPE Parameters

ASID_WIDTH ⇒ 0 to 16

Number of implemented ASID bits. Maximum is 16 for XLEN==64, and 9 for XLEN==32

HSTATEEN_ENVCFG_TYPE ⇒ [rw, read-only-0, read-only-1]

Behavior of the hstateen0.ENVCFG bit:

- 'rw': read-write
- 'read-only-0': read-only, fixed to 0
- 'read-only-1': read-only, fixed to 1

MSTATEEN_ENVCFG_TYPE ⇒ [rw, read-only-0, read-only-1]

Behavior of the mstateen0.ENVCFG bit:

- 'rw': read-write
- 'read-only-0': read-only, fixed to 0
- 'read-only-1': read-only, fixed to 1

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

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

MSTATUS_FS_WRITABLE ⇒ boolean

When S is enabled but F is not, mstatus.FS is optionally writable.

MSTATUS_TVM_IMPLEMENTED ⇒ boolean

Whether or not mstatus.TVM is implemented.

When not implemented mstatus.TVM will be read-only-zero.

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

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

MSTATUS_VS_WRITABLE ⇒ boolean

When S is enabled but V is not, mstatus.VS is optionally writable.

MUTABLE_MISA_S ⇒ boolean

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

REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION ⇒ boolean

When true, stval is written with the encoding of an instruction that causes an `IllegalInstruction` exception.

When false stval is written with 0 when an `IllegalInstruction` exception occurs.

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ⇒ boolean

When true, mtval is written with the virtual PC of an instructino when fetch causes an `InstructionPageFault`.

WHen false, mtval is written with 0 when an instruction fetch causes an `InstructionPageFault`.

REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ⇒ boolean

When true, mtval is written with the virtual address of a load when it causes a `LoadPageFault`.

WHen false, mtval is written with 0 when a load causes a `LoadPageFault`.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ⇒ boolean

When true, mtval is written with the virtual address of a store when it causes a `StoreAmoPageFault`.

WHen false, mtval is written with 0 when a store causes a `StoreAmoPageFault`.

REPORT_VA_IN_STVAL_ON_BREAKPOINT ⇒ boolean

When true, stval is written with the virtual PC of the EBREAK instruction (same information as mepc).

When false, stval is written with 0 on an EBREAK instruction.

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ⇒ boolean

When true, stval is written with the virtual PC of an instructino when fetch causes an `InstructionAccessFault`.

WHen false, stval is written with 0 when an instruction fetch causes an `InstructionAccessFault`.

REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ⇒ boolean

When true, stval is written with the virtual PC when an instruction fetch is misaligned.

When false, stval is written with 0 when an instruction fetch is misaligned.

Note that when IALIGN=16 (i.e., when the C or one of the `Zc*` extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ⇒ boolean

When true, stval is written with the virtual PC of an instructino when fetch causes an `InstructionPageFault`.

WHen false, stval is written with 0 when an instruction fetch causes an `InstructionPageFault`.

REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ⇒ boolean

When true, stval is written with the virtual address of a load when it causes a `LoadAccessFault`.

WHen false, stval is written with 0 when a load causes a `LoadAccessFault`.

REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ⇒ boolean

When true, stval is written with the virtual address of a load instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, stval is written with `0` when a load address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ⇒ boolean

When true, stval is written with the virtual address of a load when it causes a `LoadPageFault`.

WHen false, stval is written with `0` when a load causes a `LoadPageFault`.

REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ⇒ boolean

When true, stval is written with the virtual address of a store when it causes a `StoreAmoAccessFault`.

WHen false, stval is written with `0` when a store causes a `StoreAmoAccessFault`.

REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED ⇒ boolean

When true, stval is written with the virtual address of a store instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, stval is written with `0` when a store address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT ⇒ boolean

When true, stval is written with the virtual address of a store when it causes a `StoreAmoPageFault`.

WHen false, stval is written with `0` when a store causes a `StoreAmoPageFault`.

SATP_MODE_BARE ⇒ boolean

Whether or not satp.MODE == Bare is supported.

SCOUNTENABLE_EN ⇒ 32-element array of boolean

Indicates which counters can delegated via scounteren

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

SCOUNTENABLE_EN[0:2] must all be false if Zicntr is not implemented. SCOUNTENABLE_EN[3:31] must all be false if Zihpm is not implemented.

STVAL_WIDTH ⇒ ≤ 0xffffffffffffffff

The number of implemented bits in stval.

Must be greater than or equal to *max*(`PHYS_ADDR_WIDTH`, `VA_SIZE`)

STVEC_MODE_DIRECT ⇒ boolean

Whether or not stvec.MODE supports Direct (`0`).

STVEC_MODE_VECTORED ⇒ boolean

Whether or not stvec.MODE supports Vectored (1).

SXLEN ⇒ [32, 64, 3264]

Set of XLENs supported in S-mode. Can be one of:

- 32: SXLEN is always 32
- 64: SXLEN is always 64
- 3264: SXLEN can be changed (via mstatus.SXL) between 32 and 64

S_MODE_ENDIANNESS ⇒ [little, big, dynamic]

Endianness of data in S-mode. Can be one of:

- little: S-mode data is always little endian
- big: S-mode data is always big endian
- dynamic: S-mode data can be either little or big endian, depending on the CSR field mstatus.SBE

TRAP_ON_ECALL_FROM_S ⇒ boolean

Whether or not an ECALL-from-S-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY ⇒ boolean

For implementations that make satp.MODE read-only zero (always Bare, *i.e.*, no virtual translation is implemented), attempts to execute an SFENCE.VMA instruction might raise an illegal-instruction exception.

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY indicates whether or not that exception occurs.

TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY has no effect when some virtual translation mode is supported.

A.6. Extension Sm

Long Name: Machine mode
Version Requirement: ~> 1.11.0

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.

1.12.0

State

ratified

Ratification date

2021-12

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 optional 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.

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.6.1. 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.6.2. 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 Exception Return**
wfi	**Wait for interrupt**

A.6.3. 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
mepc	Machine Exception Program Counter	0x341	M
mhartid	Machine Hart ID	0xf14	M
mideleg	Machine Interrupt Delegation	0x303	M
mie	Machine Interrupt Enable	0x304	M
mimpid	Machine Implementation ID	0xf13	M
mip	Machine Interrupt Pending	0x344	M
misa	Machine ISA Control	0x301	M
mnepc	Machine Exception Program Counter	0x741	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.6.4. IN-SCOPE Parameters

ARCH_ID ⇒ 64-bit integer

Vendor-specific architecture ID in marchid

IMP_ID ⇒ 64-bit integer

Vendor-specific implementation ID in mimpid

MISALIGNED_LDST ⇒ boolean

Does the implementation perform non-atomic misaligned loads and stores to main memory (does **not** affect misaligned support to device memory)? If not, the implementation always throws a misaligned exception.

MISALIGNED_LDST_EXCEPTION_PRIORITY ⇒ [low, high]

The relative priority of a load/store/AMO exception vs. load/store/AMO page-fault or access-fault exceptions.

May be one of:

low	Misaligned load/store/AMO exceptions are always lower priority than load/store/AMO page-fault and access-fault exceptions.
high	Misaligned load/store/AMO exceptions are always higher priority than load/store/AMO page-fault and access-fault exceptions.

MISALIGNED_LDST_EXCEPTION_PRIORITY cannot be "high" when MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE is non-zero, since the atomicity of an access cannot be determined in that case until after address translation.

MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE ⇒ [0, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]

The maximum granule size, in bytes, that the hart can atomically perform a misaligned load/store/AMO without raising a Misaligned exception. When MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE is `0`, the hart cannot atomically perform a misaligned load/store/AMO. When a power of two, the hart can atomically load/store/AMO a misaligned access that is fully contained in a MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE-aligned region.

Even if the hart is capable of performing a misaligned load/store/AMO atomically, a misaligned exception may still occur if the access does not have the appropriate Misaligned Atomicity Granule PMA set.

MISALIGNED_SPLIT_STRATEGY ⇒ [by_byte, custom]

When misaligned accesses are supported, this determines the **order** in the implementation appears to process the load/store, which determines how/which exceptions will be reported

Options:

- by_byte: The load/store appears to be broken into byte-sized accesses that processed sequentially from smallest address to largest address
- custom: Something else. Will result in a call to unpredictable() in the execution

MISA_CSR_IMPLEMENTED ⇒ boolean

Whether or not the misa CSR returns zero or a non-zero value.

Possible values:

true

The misa CSR returns a non-zero value.

false

The misa CSR is read-only-`0`.

MTVAL_WIDTH ⇒ ≤ 64

The number of implemented bits in the mtval CSR. This is the CSR that may be written when a trap is taken into M-mode with exception-specific information to assist software in handling the trap (e.g., address associated with exception).

Must be greater than or equal to *max*(`PHYS_ADDR_WIDTH`, `VA_SIZE`)

MTVEC_BASE_ALIGNMENT_DIRECT ⇒ 4 to 64

Byte alignment for mtvec.BASE when mtvec.MODE is Direct.

Cannot be less than 4-byte alignment.

MTVEC_BASE_ALIGNMENT_VECTORED ⇒ ≥ 4

Byte alignment for mtvec.BASE when mtvec.MODE is Vectored.

Cannot be less than 4-byte alignment.

MTVEC_MODES ⇒ 1-element to 2-element array of [0, 1]

Modes supported by mtvec.MODE. If only one, it is assumed to be read-only with that value.

MXLEN ⇒ [32, 64]

XLEN in M-mode

M_MODE_ENDIANNESS ⇒ [little, big, dynamic]

Endianness of data in M-mode. Can be one of:

little	M-mode data is always little endian
big	M-mode data is always big endian
dynamic	M-mode data can be either little or big endian, depending on the CSR field mstatus.MBE

PHYS_ADDR_WIDTH ⇒ 1 to 64

Number of bits in the physical address space.

PRECISE_SYNCHRONOUS_EXCEPTIONS ⇒ boolean

Whether or not all synchronous exceptions are precise.

If false, any exception not otherwise mandated to precise (e.g., PMP violation) will cause execution to enter an unpredictable state.

TRAP_ON_EBREAK ⇒ boolean

Whether or not an EBREAK causes a synchronous exception.

The spec states that implementations may handle EBREAKs transparently without raising a trap, in which case the EEI must provide a builtin.

TRAP_ON_ECALL_FROM_M ⇒ boolean

Whether or not an ECALL-from-M-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

VENDOR_ID_BANK ⇒ 25-bit integer

JEDEC Vendor ID bank, for mvendorid

VENDOR_ID_OFFSET ⇒ 7-bit integer

Vendor JEDEC code offset, for mvendorid

A.6.5. OUT-OF-SCOPE Parameters

CONFIG_PTR_ADDRESS ⇒ 64-bit integer

Physical address of the unified discovery configuration data structure. This address is reported in the mconfigptr CSR.

PMA_GRANULARITY ⇒ 2 to 66

log2 of the smallest supported PMA region.

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

REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ⇒ boolean

When true, mtval is written with the encoding of an instruction that causes an `IllegalInstruction` exception.

When false mtval is written with `0` when an `IllegalInstruction` exception occurs.

REPORT_VA_IN_MTVAL_ON_BREAKPOINT ⇒ boolean

When true, mtval is written with the virtual PC of the EBREAK instruction (same information as mepc).

When 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

When true, mtval is written with the virtual PC of an instructino when fetch causes an `InstructionAccessFault`.

WHen false, mtval is written with `0` when an instruction fetch causes an `InstructionAccessFault`.

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ⇒ boolean

When true, mtval is written with the virtual PC when an instruction fetch is misaligned.

When false, mtval is written with `0` when an instruction fetch is misaligned.

Note that when IALIGN=16 (i.e., when the C or one of the `Zc*` extensions are implemented), it is impossible to generate a misaligned fetch, and so this parameter has no effect.

REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ⇒ boolean

When true, mtval is written with the virtual address of a load when it causes a `LoadAccessFault`.

WHen false, mtval is written with `0` when a load causes a `LoadAccessFault`.

REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ⇒ boolean

When true, mtval is written with the virtual address of a load instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, mtval is written with `0` when a load address is misaligned and MISALIGNED_LDST is false.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ⇒ boolean

When true, mtval is written with the virtual address of a store when it causes a `StoreAmoAccessFault`.

WHen false, mtval is written with `0` when a store causes a `StoreAmoAccessFault`.

REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ⇒ boolean

When true, mtval is written with the virtual address of a store instruction when the address is misaligned and MISALIGNED_LDST is false.

When false, mtval is written with `0` when a store address is misaligned and MISALIGNED_LDST is false.

TRAP_ON_ILLEGAL_WLRL ⇒ boolean

When true, writing an illegal value to a WLRL CSR field raises an `IllegalInstruction` exception.

When false, writing an illegal value to a WLRL CSR field is `unpredictable`.

TRAP_ON_RESERVED_INSTRUCTION ⇒ boolean

When true, fetching an unimplemented and/or undefined instruction from the standard/reserved encoding space will cause an `IllegalInstruction` exception.

When false, fetching such an instruction is `UNPREDICTABLE`.

TRAP_ON_UNIMPLEMENTED_CSR ⇒ boolean

When true, accessing an unimplemented CSR (via a Zicsr instruction) will cause an `IllegalInstruction` exception.

When false, accessing an unimplemented CSR (via a Zicsr instruction) is `unpredictable`.

TRAP_ON_UNIMPLEMENTED_INSTRUCTION ⇒ boolean

When true, fetching an unimplemented instruction from the custom encoding space will cause an `IllegalInstruction` exception.

When false, fetching an unimplemented instruction is `UNPREDICTABLE`.

A.7. Extension Smpmp

Long Name: Physical Memory Protection
Version Requirement: ~> 1.11.0

1.11.0

State

ratified

Ratification date

2019-12

Changes

- Made the unused `pmpaddr` and `pmpcfg` fields **WARL**, rather than **WIRI**.
- Specified semantics for PMP regions coarser than four bytes.

1.12.0

State

ratified

Ratification date

2021-12

Changes

- PMP changes require an SFENCE.VMA on any hart that implements page-based virtual memory, even if VM is not currently enabled.
- PMP reset values are now platform-defined.
- An additional 48 optional PMP registers have been defined.

1.13.0

State

frozen

A.7.1. Synopsis

To support secure processing and contain faults, it is desirable to limit the physical addresses accessible by software running on a hart. The optional PMP (Physical Memory Protection) unit provides per-hart machine-mode control registers to allow physical memory access privileges (read, write, execute) to be specified for each physical memory region. The PMP values are checked in parallel with the PMA checks.

This is a placeholder extension name defined by the `riscv-unified-db` and isn't considered an extension in the Privileged ISA manual. This was chosen as an extension rather than a M-mode parameter since the PMP has no visible ISA features (such as read-only-0 CSRs) if not present in an implementation. Making it an extension in the database prevents having the PMP CSRs show up in implementations that don't have a PMP.

A.7.2. CSRs

The following 80 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
pmpaddr0	PMP Address 0	0x3b0	M
pmpaddr1	PMP Address 1	0x3b1	M
pmpaddr10	PMP Address 10	0x3ba	M
pmpaddr11	PMP Address 11	0x3bb	M
pmpaddr12	PMP Address 12	0x3bc	M
pmpaddr13	PMP Address 13	0x3bd	M
pmpaddr14	PMP Address 14	0x3be	M
pmpaddr15	PMP Address 15	0x3bf	M
pmpaddr16	PMP Address 16	0x3c0	M
pmpaddr17	PMP Address 17	0x3c1	M
pmpaddr18	PMP Address 18	0x3c2	M
pmpaddr19	PMP Address 19	0x3c3	M
pmpaddr2	PMP Address 2	0x3b2	M
pmpaddr20	PMP Address 20	0x3c4	M
pmpaddr21	PMP Address 21	0x3c5	M
pmpaddr22	PMP Address 22	0x3c6	M
pmpaddr23	PMP Address 23	0x3c7	M
pmpaddr24	PMP Address 24	0x3c8	M
pmpaddr25	PMP Address 25	0x3c9	M
pmpaddr26	PMP Address 26	0x3ca	M
pmpaddr27	PMP Address 27	0x3cb	M
pmpaddr28	PMP Address 28	0x3cc	M
pmpaddr29	PMP Address 29	0x3cd	M
pmpaddr3	PMP Address 3	0x3b3	M
pmpaddr30	PMP Address 30	0x3ce	M
pmpaddr31	PMP Address 31	0x3cf	M
pmpaddr32	PMP Address 32	0x3d0	M
pmpaddr33	PMP Address 33	0x3d1	M
pmpaddr34	PMP Address 34	0x3d2	M
pmpaddr35	PMP Address 35	0x3d3	M
pmpaddr36	PMP Address 36	0x3d4	M
pmpaddr37	PMP Address 37	0x3d5	M
pmpaddr38	PMP Address 38	0x3d6	M
pmpaddr39	PMP Address 39	0x3d7	M
pmpaddr4	PMP Address 4	0x3b4	M
pmpaddr40	PMP Address 40	0x3d8	M
pmpaddr41	PMP Address 41	0x3d9	M
pmpaddr42	PMP Address 42	0x3da	M
pmpaddr43	PMP Address 43	0x3db	M
pmpaddr44	PMP Address 44	0x3dc	M
pmpaddr45	PMP Address 45	0x3dd	M
pmpaddr46	PMP Address 46	0x3de	M
pmpaddr47	PMP Address 47	0x3df	M
pmpaddr48	PMP Address 48	0x3e0	M
pmpaddr49	PMP Address 49	0x3e1	M

Name	Long Name	Address	Mode
pmpaddr5	PMP Address 5	0x3b5	M
pmpaddr50	PMP Address 50	0x3e2	M
pmpaddr51	PMP Address 51	0x3e3	M
pmpaddr52	PMP Address 52	0x3e4	M
pmpaddr53	PMP Address 53	0x3e5	M
pmpaddr54	PMP Address 54	0x3e6	M
pmpaddr55	PMP Address 55	0x3e7	M
pmpaddr56	PMP Address 56	0x3e8	M
pmpaddr57	PMP Address 57	0x3e9	M
pmpaddr58	PMP Address 58	0x3ea	M
pmpaddr59	PMP Address 59	0x3eb	M
pmpaddr6	PMP Address 6	0x3b6	M
pmpaddr60	PMP Address 60	0x3ec	M
pmpaddr61	PMP Address 61	0x3ed	M
pmpaddr62	PMP Address 62	0x3ee	M
pmpaddr63	PMP Address 63	0x3ef	M
pmpaddr7	PMP Address 7	0x3b7	M
pmpaddr8	PMP Address 8	0x3b8	M
pmpaddr9	PMP Address 9	0x3b9	M
pmpcfg0	PMP Configuration Register 0	0x3a0	M
pmpcfg1	PMP Configuration Register 1	0x3a1	M
pmpcfg10	PMP Configuration Register 10	0x3aa	M
pmpcfg11	PMP Configuration Register 11	0x3ab	M
pmpcfg12	PMP Configuration Register 12	0x3ac	M
pmpcfg13	PMP Configuration Register 13	0x3ad	M
pmpcfg14	PMP Configuration Register 14	0x3ae	M
pmpcfg15	PMP Configuration Register 15	0x3af	M
pmpcfg2	PMP Configuration Register 2	0x3a2	M
pmpcfg3	PMP Configuration Register 3	0x3a3	M
pmpcfg4	PMP Configuration Register 4	0x3a4	M
pmpcfg5	PMP Configuration Register 5	0x3a5	M
pmpcfg6	PMP Configuration Register 6	0x3a6	M
pmpcfg7	PMP Configuration Register 7	0x3a7	M
pmpcfg8	PMP Configuration Register 8	0x3a8	M
pmpcfg9	PMP Configuration Register 9	0x3a9	M

A.7.3. OUT-OF-SCOPE Parameters

NUM_PMP_ENTRIES ⇒ 0 to 64

Number of implemented PMP entries. Can be any value between 0-64, inclusive.

The architecture mandates that the number of implemented PMP registers must appear to be 0, 16, or 64.

Therefore, pmp registers will behave as follows according to NUN_PMP_ENTRIES:

NUM_PMP_ENTRIES	pmpaddr<0-15> / pmpcfg<0-3>	pmpaddr<16-63> / pmpcfg<4-15>
0	N	N
1-16	Y	N
17-64	Y	Y

- N = Not implemented; access will cause `IllegalInstruction` if TRAP_ON_UNIMPLEMENTED_CSR is true
- Y = Implemented; access will not cause an exception (from M-mode), but register may be read-only-zero if NUM_PMP_ENTRIES is less than the corresponding register

> `pmpcfgN` *for an odd N never exists when XLEN == 64*

When NUM_PMP_ENTRIES is not exactly 0, 16, or 64, some extant pmp registers, and associated pmpNcfg, will be read-only zero (but will never cause

an exception).

PMP_GRANULARITY ⇒ 2 to 66

log2 of the smallest supported PMP region.

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

Note that PMP_GRANULARITY is equal to G+2 (not G) as described in the privileged architecture.

A.8. Extension U

Long Name: User-mode privilege level
Version Requirement: ~> 1.0.0

1.0.0

State

ratified

Ratification date

2019-12

A.8.1. Synopsis

User-mode privilege level is supported by an implementation if the U extension is present. Note that the RISC-V ISA doesn't formally define a U extension and it is only discussed in the Privileged ISA manual.

A.8.2. CSRs

The following 4 CSRs are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
mcounteren	Machine Counter Enable	0x306	M
menvcfg	Machine Environment Configuration	0x30a	M
menvcfgh	Machine Environment Configuration	0x31a	M
senvcfg	Supervisor Environment Configuration	0x10a	S

A.8.3. OUT-OF-SCOPE Parameters

MUTABLE_MISA_U ⇒ boolean

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

TRAP_ON_ECALL_FROM_U ⇒ boolean

Whether or not an ECALL-from-U-mode causes a synchronous exception.

The spec states that implementations may handle ECALLs transparently without raising a trap, in which case the EEI must provide a builtin.

UXLEN ⇒ [32, 64, 3264]

Set of XLENs supported in U-mode. Can be one of:

- 32: SXLEN is always 32
- 64: SXLEN is always 64
- 3264: SXLEN can be changed (via mstatus.UXL) between 32 and 64

U_MODE_ENDIANNESS ⇒ [little, big, dynamic]

Endianness of data in U-mode. Can be one of:

- little: U-mode data is always little endian
- big: U-mode data is always big endian
- dynamic: U-mode data can be either little or big endian, depending on the CSR field mstatus.UBE

A.9. Extension Zce

Long Name: Compressed instructions for microcontrollers
Version Requirement: ~> 1.0.0

1.0.0

State

ratified

Ratification date

2023-04

Implies

- Zca version 1.0.0
- Zcb version 1.0.0
- Zcmp version 1.0.0
- Zcmt version 1.0.0

A.9.1. Synopsis

The Zce extension is intended to be used for microcontrollers, and includes all relevant Zc extensions.

- Specifying Zce on RV32 without F includes Zca, Zcb, Zcmp, Zcmt
- Specifying Zce on RV32 with F includes Zca, Zcb, Zcmp, Zcmt and Zcf
- Specifying Zce on RV64 always includes Zca, Zcb, Zcmp, Zcmt
- Zcf doesn't exist for RV64

Therefore common ISA strings can be updated as follows to include the relevant Zc extensions, for example:

- RV32IMC becomes RV32IM_Zce
- RV32IMCF becomes RV32IMF_Zce

A.10. Extension Zicntr

Long Name: Architectural performance counters
Version Requirement: ~> 2.0

2.0.0

State

ratified

Ratification date

2019-12

A.10.1. Synopsis

Architectural performance counters

A.10.2. CSRs

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

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

A.10.3. IN-SCOPE Parameters

TIME_CSR_IMPLEMENTED ⇒ boolean

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

Possible values:

true

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

false

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

A.11. Extension Zicsr

Long Name: Control and status registers
Version Requirement: ~> 2.0

2.0.0

State

ratified

A.11.1. Synopsis

Control and status registers

A.11.2. 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

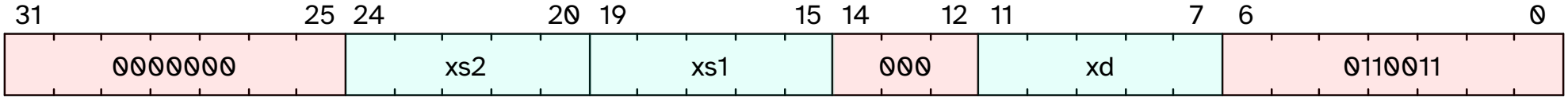
B.1. add

Integer add

This instruction is defined by:

- I, version >= I@2.1.0

B.1.1. Encoding



B.1.2. Description

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

B.1.3. Access

M	S	U
Always	Always	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.

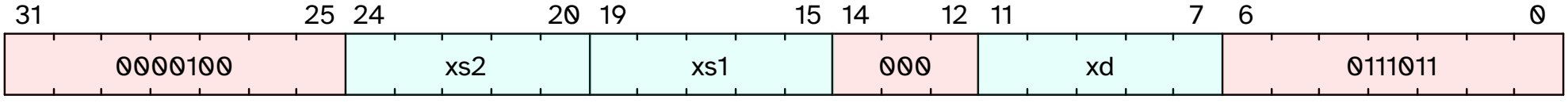
B.2. add.uw

Add unsigned word

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.2.1. Encoding



B.2.2. Description

Performs an XLEN-wide addition between rs2 and the zero-extended least-significant word of rs1.

B.2.3. Access

M	S	U
Always	Always	Always

B.2.4. Decode Variables

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

B.2.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[xd] = X[xs2] + X[xs1][31:0];
```

B.2.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_ADDUW    => 0b00,
    RISCV_SH1ADDUW => 0b01,
    RISCV_SH2ADDUW => 0b10,
    RISCV_SH3ADDUW => 0b11
  };
  let result : xlenbits = (zero_extend(rs1_val[31..0]) << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.2.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

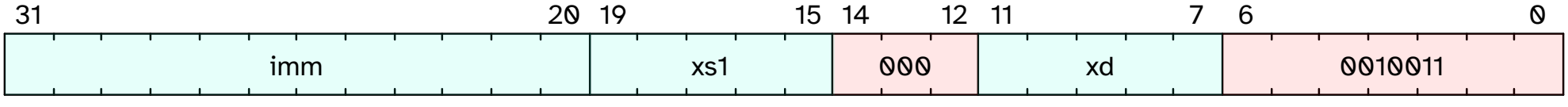
B.3. addi

Add immediate

This instruction is defined by:

- I, version >= I@2.1.0

B.3.1. Encoding



B.3.2. Description

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

B.3.3. Access

M	S	U
Always	Always	Always

B.3.4. Decode Variables

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

B.3.5. IDL Operation

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

B.3.6. 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.3.7. Exceptions

This instruction does not generate synchronous exceptions.

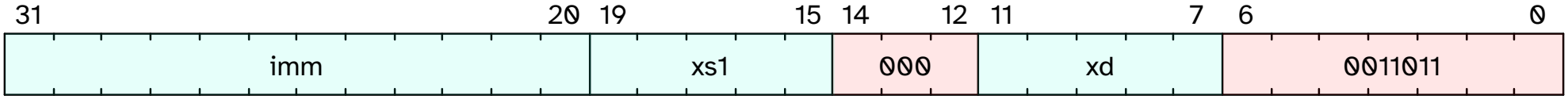
B.4. addiw

Add immediate word

This instruction is defined by:

- I, version >= I@2.1.0

B.4.1. Encoding



B.4.2. Description

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

B.4.3. Access

M	S	U
Always	Always	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

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

B.4.6. Sail Operation

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

B.4.7. Exceptions

This instruction does not generate synchronous exceptions.

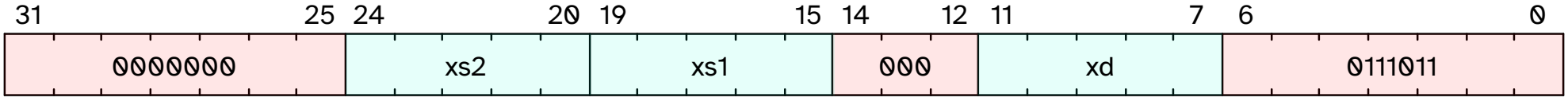
B.5. addw

Add word

This instruction is defined by:

- I, version >= I@2.1.0

B.5.1. Encoding



B.5.2. Description

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

B.5.3. Access

M	S	U
Always	Always	Always

B.5.4. Decode Variables

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

B.5.5. IDL Operation

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

B.5.6. Sail Operation

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

B.5.7. Exceptions

This instruction does not generate synchronous exceptions.

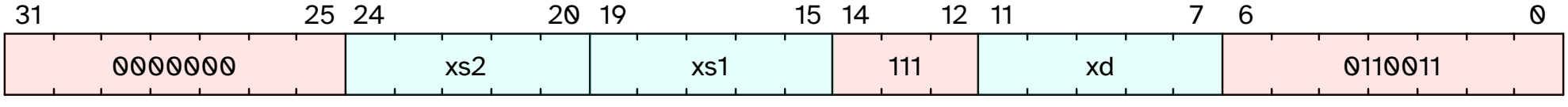
B.6. and

And

This instruction is defined by:

- I, version >= I@2.1.0

B.6.1. Encoding



B.6.2. Description

And xs1 with xs2, and store the result in xd

B.6.3. Access

M	S	U
Always	Always	Always

B.6.4. Decode Variables

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

B.6.5. IDL Operation

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

B.6.6. 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.6.7. Exceptions

This instruction does not generate synchronous exceptions.

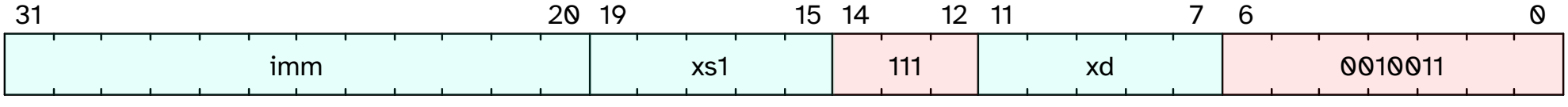
B.7. andi

And immediate

This instruction is defined by:

- I, version >= I@2.1.0

B.7.1. Encoding



B.7.2. Description

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

B.7.3. Access

M	S	U
Always	Always	Always

B.7.4. Decode Variables

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

B.7.5. IDL Operation

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

B.7.6. 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.7.7. Exceptions

This instruction does not generate synchronous exceptions.

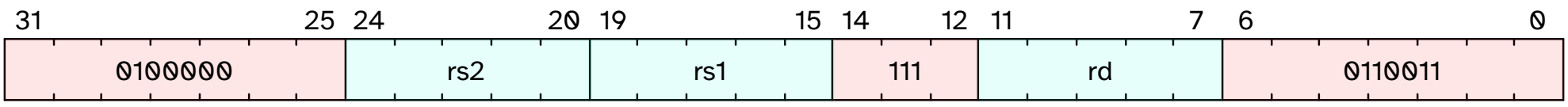
B.8. andn

AND with inverted operand

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.8.1. Encoding



B.8.2. Description

Performs the bitwise logical AND operation between `rs1` and the bitwise inversion of `rs2`.

B.8.3. Access

M	S	U
Always	Always	Always

B.8.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.8.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] & ~X[rs1];
```

B.8.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.8.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

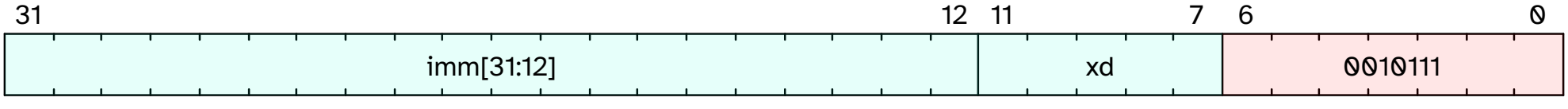
B.9. auipc

Add upper immediate to pc

This instruction is defined by:

- I, version >= I@2.1.0

B.9.1. Encoding



B.9.2. Description

Add an immediate to the current PC.

B.9.3. Access

M	S	U
Always	Always	Always

B.9.4. Decode Variables

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

B.9.5. IDL Operation

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

B.9.6. 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.9.7. Exceptions

This instruction does not generate synchronous exceptions.

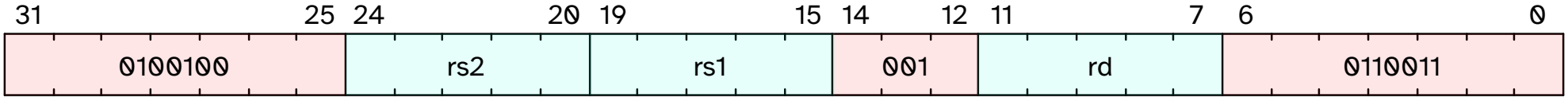
B.10. bclr

Single-Bit clear (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

B.10.1. Encoding



B.10.2. Description

Returns rs1 with a single bit cleared at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

B.10.3. Access

M	S	U
Always	Always	Always

B.10.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.10.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = X[rs1] & ~(1 << index);
```

B.10.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << rs2_val[4..0]
                        else zero_extend(0b1) << rs2_val[5..0];
  let result : xlenbits = match op {
    RISCV_BCLR => rs1_val & ~(mask),
    RISCV_BEXT => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
    RISCV_BINV => rs1_val ^ mask,
    RISCV_BSET => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.10.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.11. bclri

Single-Bit clear (Immediate)

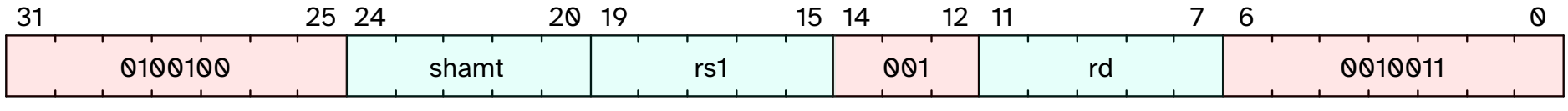
This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

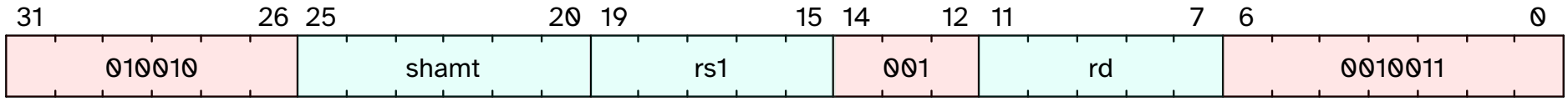
B.11.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.11.2. Description

Returns rs1 with a single bit cleared at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.11.3. Access

M	S	U
Always	Always	Always

B.11.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.11.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[rd] = X[rs1] & ~(1 << index);
```

B.11.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << shamt[4..0]
                        else zero_extend(0b1) << shamt;
  let result : xlenbits = match op {
    RISCV_BCLRI => rs1_val & ~(mask),
    RISCV_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
```

```
    RISCV_BINVI => rs1_val ^ mask,
    RISCV_BSETI => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.11.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

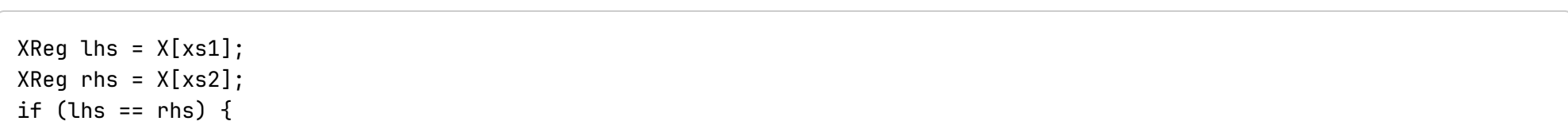
B.12. beq

Branch if equal

This instruction is defined by:

- I, version >= I@2.1.0

B.12.1. Encoding



B.12.2. 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.12.3. Access

M	S	U
Always	Always	Always

B.12.4. 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.12.5. IDL Operation

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

B.12.6. 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.12.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

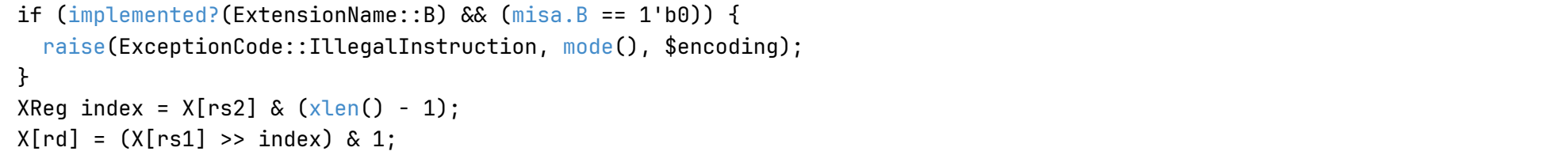
B.13. bext

Single-Bit extract (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

B.13.1. Encoding



B.13.2. Description

Returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

B.13.3. Access

M	S	U
Always	Always	Always

B.13.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.13.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = (X[rs1] >> index) & 1;
```

B.13.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << rs2_val[4..0]
                        else zero_extend(0b1) << rs2_val[5..0];
  let result : xlenbits = match op {
    RISCV_BCLR => rs1_val & ~(mask),
    RISCV_BEXT => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
    RISCV_BINV => rs1_val ^ mask,
    RISCV_BSET => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.13.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.14. bexti

Single-Bit extract (Immediate)

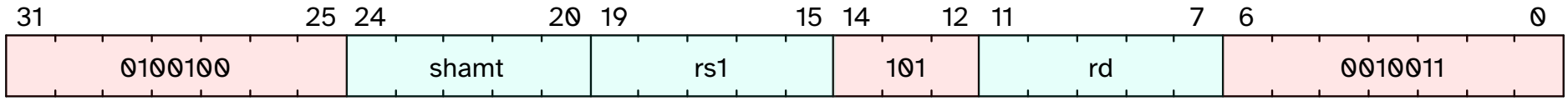
This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

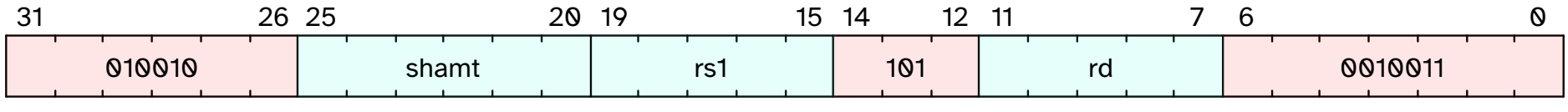
B.14.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.14.2. Description

Returns a single bit extracted from rs1 at the index specified in rs2. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.14.3. Access

M	S	U
Always	Always	Always

B.14.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.14.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[rd] = (X[rs1] >> index) & 1;
```

B.14.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << shamt[4..0]
                        else zero_extend(0b1) << shamt;
  let result : xlenbits = match op {
    RISCV_BCLRI => rs1_val & ~(mask),
    RISCV_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
```

```
    RISCV_BINVI => rs1_val ^ mask,
    RISCV_BSETI => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.14.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

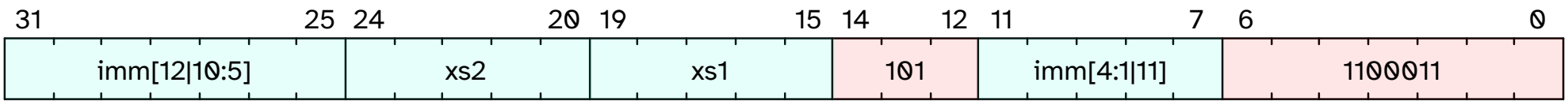
B.15. bge

Branch if greater than or equal

This instruction is defined by:

- I, version >= I@2.1.0

B.15.1. Encoding



B.15.2. 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.15.3. Access

M	S	U
Always	Always	Always

B.15.4. 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.15.5. IDL Operation

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

B.15.6. 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.15.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

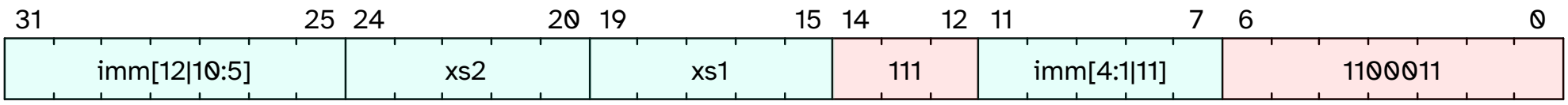
B.16. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.16.1. Encoding



B.16.2. 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.16.3. Access

M	S	U
Always	Always	Always

B.16.4. 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.16.5. IDL Operation

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

B.16.6. 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.16.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

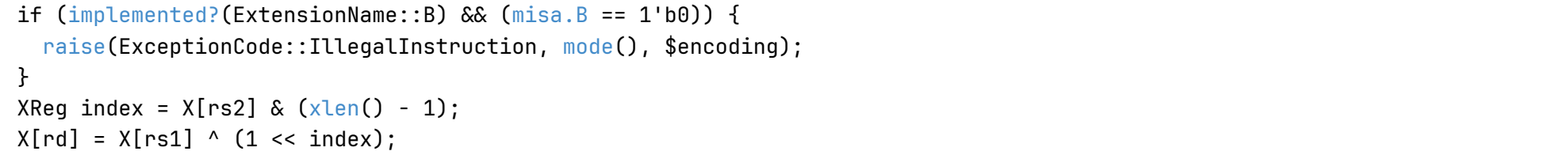
B.17. binv

Single-Bit invert (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

B.17.1. Encoding



B.17.2. Description

Returns rs1 with a single bit inverted at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

B.17.3. Access

M	S	U
Always	Always	Always

B.17.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.17.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = X[rs1] ^ (1 << index);
```

B.17.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << rs2_val[4..0]
                        else zero_extend(0b1) << rs2_val[5..0];
  let result : xlenbits = match op {
    RISCV_BCLR => rs1_val & ~(mask),
    RISCV_BEXT => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
    RISCV_BINV => rs1_val ^ mask,
    RISCV_BSET => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.17.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.18. binvi

Single-Bit invert (Immediate)

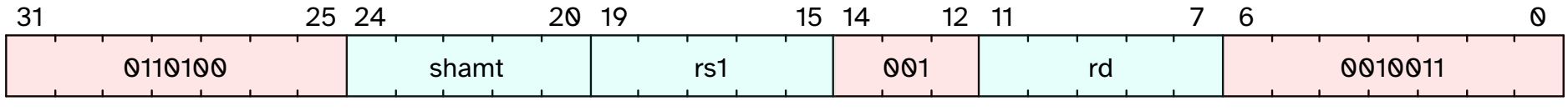
This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

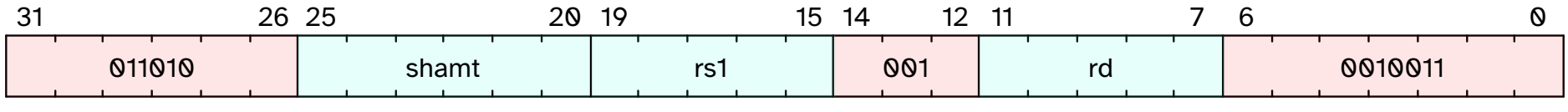
B.18.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.18.2. Description

Returns rs1 with a single bit inverted at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.18.3. Access

M	S	U
Always	Always	Always

B.18.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.18.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[rd] = X[rs1] ^ (1 << index);
```

B.18.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << shamt[4..0]
                        else zero_extend(0b1) << shamt;
  let result : xlenbits = match op {
    RISCV_BCLRI => rs1_val & ~(mask),
    RISCV_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
```

```
    RISCV_BINVI => rs1_val ^ mask,
    RISCV_BSETI => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.18.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

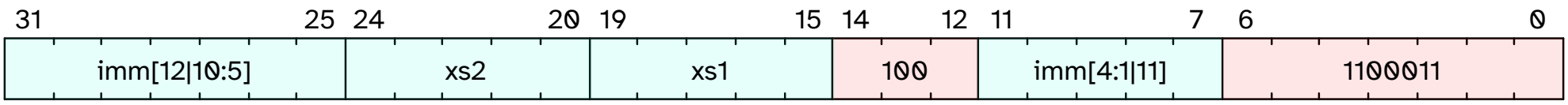
B.19. blt

Branch if less than

This instruction is defined by:

- I, version >= I@2.1.0

B.19.1. Encoding



B.19.2. 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.19.3. Access

M	S	U
Always	Always	Always

B.19.4. 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.19.5. IDL Operation

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

B.19.6. 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.19.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

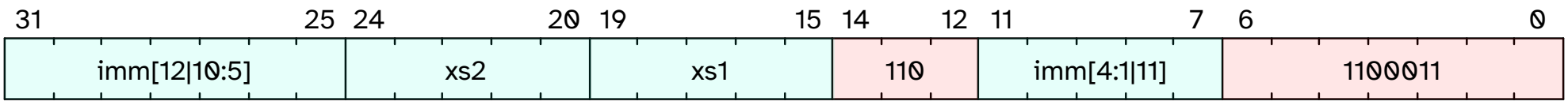
B.20. bltu

Branch if less than unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.20.1. Encoding



B.20.2. 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.20.3. Access

M	S	U
Always	Always	Always

B.20.4. 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.20.5. IDL Operation

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

B.20.6. 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.20.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

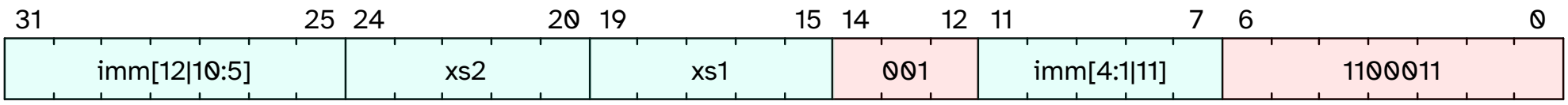
B.21. bne

Branch if not equal

This instruction is defined by:

- `I`, version >= `I@2.1.0`

B.21.1. Encoding



B.21.2. 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.21.3. Access

M	S	U
Always	Always	Always

B.21.4. 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.21.5. IDL Operation

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

B.21.6. 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.21.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

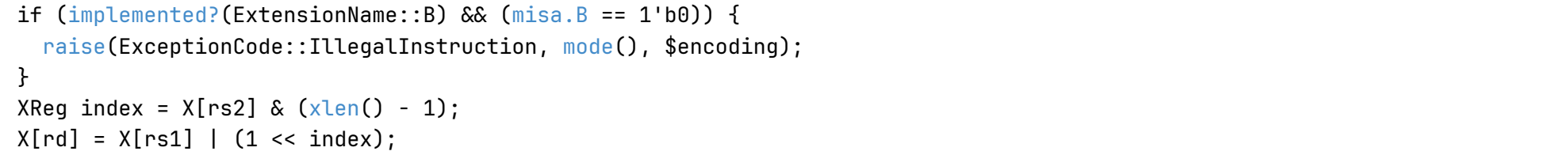
B.22. bset

Single-Bit set (Register)

This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

B.22.1. Encoding



B.22.2. Description

Returns rs1 with a single bit set at the index specified in rs2. The index is read from the lower log2(XLEN) bits of rs2.

B.22.3. Access

M	S	U
Always	Always	Always

B.22.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.22.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[rs2] & (xlen() - 1);
X[rd] = X[rs1] | (1 << index);
```

B.22.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << rs2_val[4..0]
                        else zero_extend(0b1) << rs2_val[5..0];
  let result : xlenbits = match op {
    RISCV_BCLR => rs1_val & ~(mask),
    RISCV_BEXT => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
    RISCV_BINV => rs1_val ^ mask,
    RISCV_BSET => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.22.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.23. bseti

Single-Bit set (Immediate)

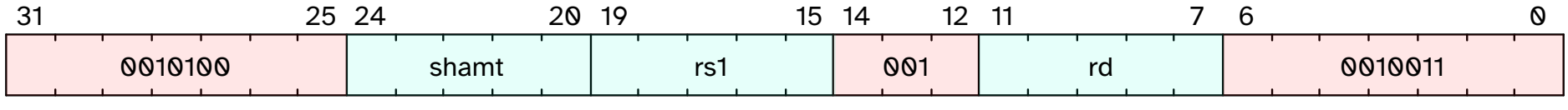
This instruction is defined by:

- Zbs, version >= Zbs@1.0.0

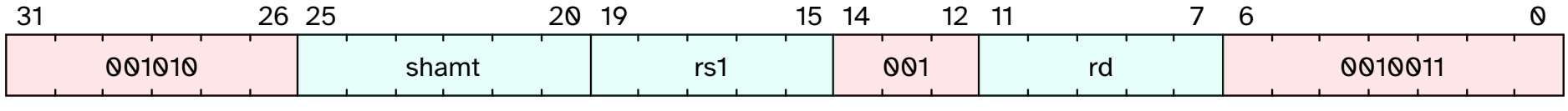
B.23.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.23.2. Description

Returns rs1 with a single bit set at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.23.3. Access

M	S	U
Always	Always	Always

B.23.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.23.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[rd] = X[rs1] | (1 << index);
```

B.23.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let mask : xlenbits = if sizeof(xlen) == 32
                        then zero_extend(0b1) << shamt[4..0]
                        else zero_extend(0b1) << shamt;
  let result : xlenbits = match op {
    RISCV_BCLRI => rs1_val & ~(mask),
    RISCV_BEXTI => zero_extend(bool_to_bits((rs1_val & mask) != zeros())),
```

```
    RISCV_BINVI => rs1_val ^ mask,
    RISCV_BSETI => rs1_val | mask
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.23.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

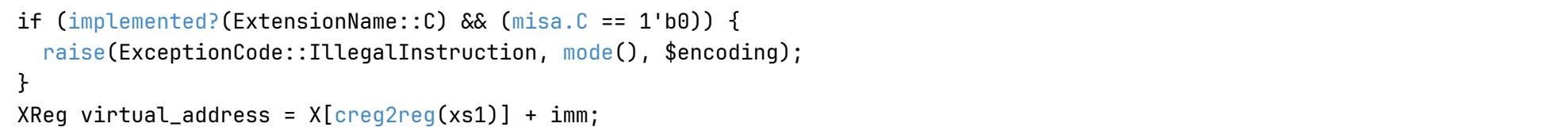
B.24. c.fld

Load double-precision

This instruction is defined by:

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

B.24.1. Encoding



B.24.2. Description

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

B.24.3. Access

M	S	U
Always	Always	Always

B.24.4. Decode Variables

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

B.24.5. IDL Operation

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

B.24.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

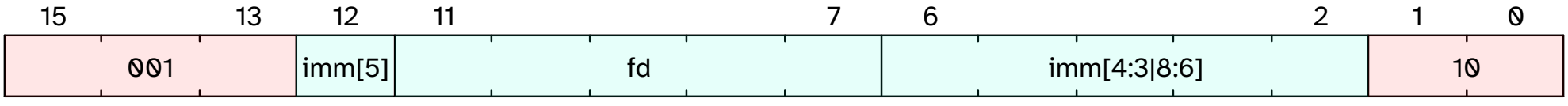
B.25. c.fldsp

Load doubleword into floating-point register from stack

This instruction is defined by:

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

B.25.1. Encoding



B.25.2. Description

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

B.25.3. Access

M	S	U
Always	Always	Always

B.25.4. Decode Variables

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

B.25.5. IDL Operation

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

B.25.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

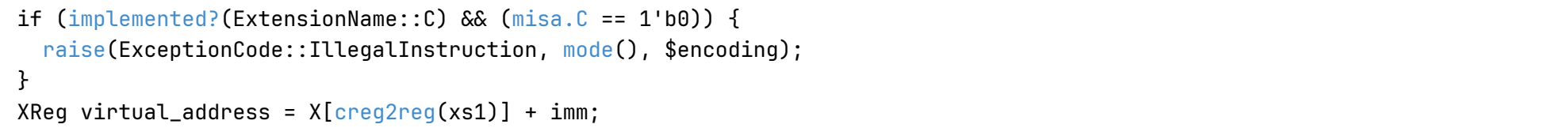
B.26. c.flw

Load single-precision

This instruction is defined by:

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

B.26.1. Encoding



B.26.2. Description

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

B.26.3. Access

M	S	U
Always	Always	Always

B.26.4. Decode Variables

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

B.26.5. IDL Operation

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

B.26.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

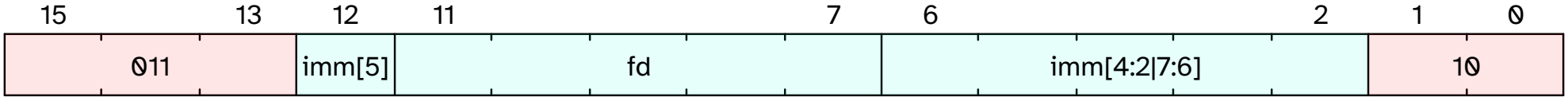
B.27. c.flwsp

Load word into floating-point register from stack

This instruction is defined by:

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

B.27.1. Encoding



B.27.2. Description

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

B.27.3. Access

M	S	U
Always	Always	Always

B.27.4. Decode Variables

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

B.27.5. IDL Operation

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

B.27.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

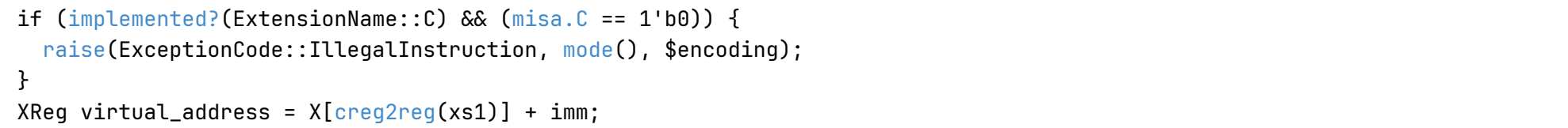
B.28. c.fsd

Store double-precision

This instruction is defined by:

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

B.28.1. Encoding



B.28.2. Description

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

B.28.3. Access

M	S	U
Always	Always	Always

B.28.4. Decode Variables

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

B.28.5. IDL Operation

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

B.28.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

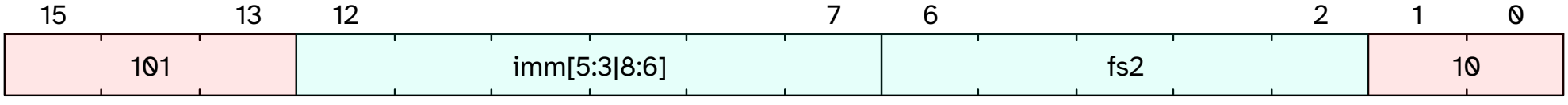
B.29. c.fsdsp

Store double-precision value to stack

This instruction is defined by:

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

B.29.1. Encoding



B.29.2. Description

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

B.29.3. Access

M	S	U
Always	Always	Always

B.29.4. Decode Variables

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

B.29.5. IDL Operation

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

B.29.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

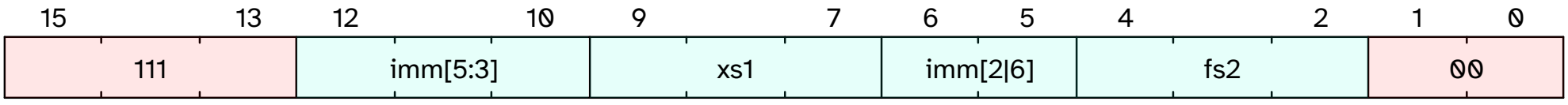
B.30. c.fsw

Store single-precision

This instruction is defined by:

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

B.30.1. Encoding



B.30.2. Description

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

B.30.3. Access

M	S	U
Always	Always	Always

B.30.4. Decode Variables

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

B.30.5. IDL Operation

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

B.30.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

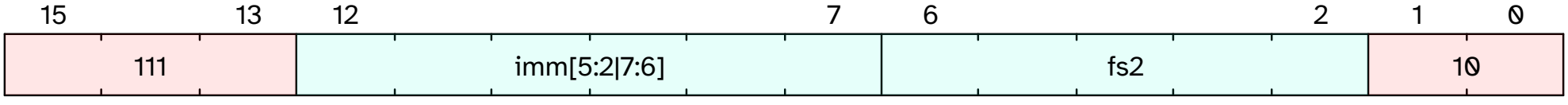
B.31. c.fswsp

Store single-precision value to stack

This instruction is defined by:

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

B.31.1. Encoding



B.31.2. Description

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

B.31.3. Access

M	S	U
Always	Always	Always

B.31.4. Decode Variables

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

B.31.5. IDL Operation

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

B.31.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

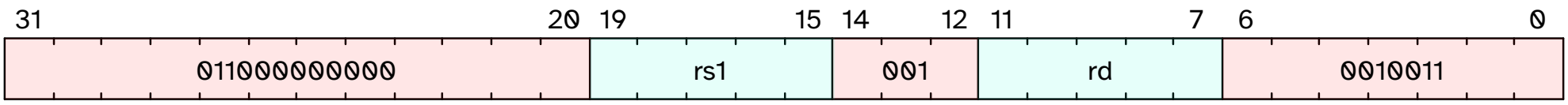
B.32. clz

Count leading zero bits

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.32.1. Encoding



B.32.2. Description

Counts the number of 0's before the first 1, starting at the most-significant bit (i.e., XLEN-1) and progressing to bit 0. Accordingly, if the input is 0, the output is XLEN, and if the most-significant bit of the input is a 1, the output is 0.

B.32.3. Access

M	S	U
Always	Always	Always

B.32.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.32.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = (xlen() - 1) - $signed(highest_set_bit(X[rs1]));
```

B.32.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : nat = 0;
  done : bool = false;
  foreach (i from (sizeof(xlen) - 1) downto 0)
    if not(done) then if rs1_val[i] == bitzero
                      then result = result + 1
                      else done = true;
  X(rd) = to_bits(sizeof(xlen), result);
  RETIRE_SUCCESS
}
```

B.32.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

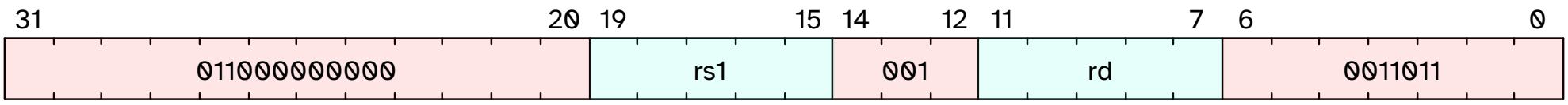
B.33. clzw

Count leading zero bits in word

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.33.1. Encoding



B.33.2. Description

Counts the number of 0's before the first 1 starting at bit 31 and progressing to bit 0. Accordingly, if the least-significant word is 0, the output is 32, and if the most-significant bit of the word (*i.e.*, bit 31) is a 1, the output is 0.

B.33.3. Access

M	S	U
Always	Always	Always

B.33.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.33.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = 31 - $signed(highest_set_bit(X[rs1][31:0]));
```

B.33.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : nat = 0;
  done : bool = false;
  foreach (i from 31 downto 0)
    if not(done) then if rs1_val[i] == bitzero
                      then result = result + 1
                      else done = true;
  X(rd) = to_bits(sizeof(xlen), result);
  RETIRE_SUCCESS
}
```

B.33.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

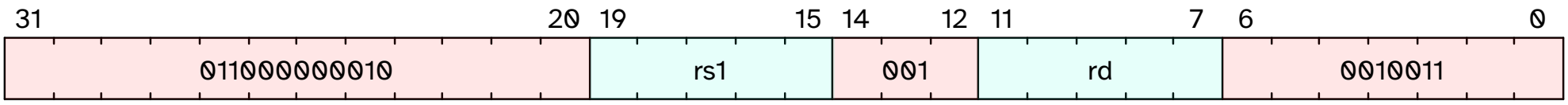
B.34. cpop

Count set bits

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.34.1. Encoding



B.34.2. Description

Counts the number of 1's (i.e., set bits) in the source register.

```
This operations is known as population count, popcount, sideways sum,
bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` is
implemented by cpop on RV32 and by cpopw on RV64. The GCC builtin
function `__builtin_popcountl (unsigned long x)` for LP64 is
implemented by cpop on RV64.
```

Listing 1. Software Hint

B.34.3. Access

M	S	U
Always	Always	Always

B.34.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.34.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg bitcount = 0;
XReg rs1_val = X[rs1];
for (U32 i = 0; i < xlen(); i++) {
  if (rs1_val[i] == 1'b1) {
    bitcount = bitcount + 1;
  }
}
X[rd] = bitcount;
```

B.34.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : nat = 0;
  foreach (i from 0 to (xlen_val - 1))
    if rs1_val[i] == bitone then result = result + 1;
  X(rd) = to_bits(sizeof(xlen), result);
  RETIRE_SUCCESS
}
```

B.34.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

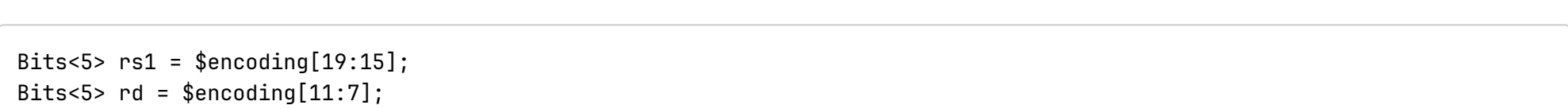
B.35. cpopw

Count set bits in word

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.35.1. Encoding



B.35.2. Description

Counts the number of 1's (i.e., set bits) in the least-significant word of the source register.

```
This operations is known as population count, popcount, sideways sum,
bit summation, or Hamming weight.

The GCC builtin function `__builtin_popcount (unsigned int x)` is
implemented by cpop on RV32 and by cpopw on RV64. The GCC builtin
function `__builtin_popcountl (unsigned long x)` for LP64 is
implemented by cpop on RV64.
```

Listing 2. Software Hint

B.35.3. Access

M	S	U
Always	Always	Always

B.35.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.35.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg bitcount = 0;
XReg rs1_val = X[rs1];
for (U32 i = 0; i < 32; i++) {
  if (rs1_val[i] == 1'b1) {
    bitcount = bitcount + 1;
  }
}
X[rd] = bitcount;
```

B.35.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : nat = 0;
  foreach (i from 0 to 31)
    if rs1_val[i] == bitone then result = result + 1;
  X(rd) = to_bits(sizeof(xlen), result);
  RETIRE_SUCCESS
}
```

B.35.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

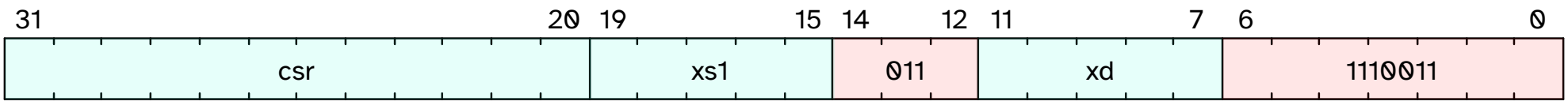
B.36. csrrc

Atomic Read and Clear Bits in CSR

This instruction is defined by:

- Zicsr, version >= Zicsr@2.0.0

B.36.1. Encoding



B.36.2. 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 `rd`. The initial value in integer register `rs1` is treated as a bit mask that specifies bit positions to be cleared in the CSR. Any bit that is high in `rs1` will cause the corresponding bit to be cleared in the CSR, if that CSR bit is writable.

For CSRRC, if `rs1=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 `rs1` and `rd` fields. Note that if `rs1` 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.36.3. Access

M	S	U
Always	Always	Always

B.36.4. Decode Variables

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

B.36.5. 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.36.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

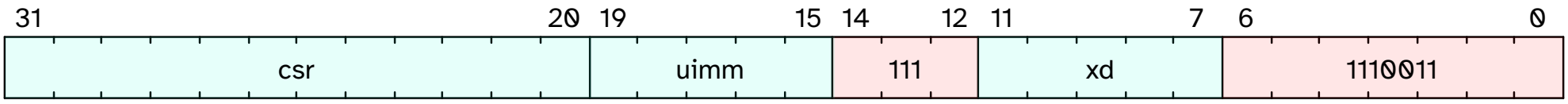
B.37. csrrci

Atomic Read and Clear Bits in CSR with Immediate

This instruction is defined by:

- Zicsr, version >= Zicsr@2.0.0

B.37.1. Encoding



B.37.2. 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 (uimm[4:0]) field encoded in the `rs1` field instead of a value from an integer register. For CSRRCI, if the `uimm[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 `rd` and `rs1` fields.

B.37.3. Access

M	S	U
Always	Always	Always

B.37.4. Decode Variables

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

B.37.5. IDL Operation

```
Boolean will_write = uimm != 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 = uimm;
  csr_sw_write(csr_handle, initial_csr_value & ~mask);
}
X[xd] = initial_csr_value;
```

B.37.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

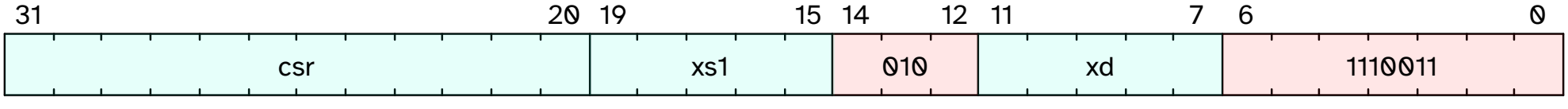
B.38. csrrs

Atomic Read and Set Bits in CSR

This instruction is defined by:

- Zicsr, version >= Zicsr@2.0.0

B.38.1. Encoding



B.38.2. 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.38.3. Access

M	S	U
Always	Always	Always

B.38.4. Decode Variables

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

B.38.5. 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.38.6. 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.38.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

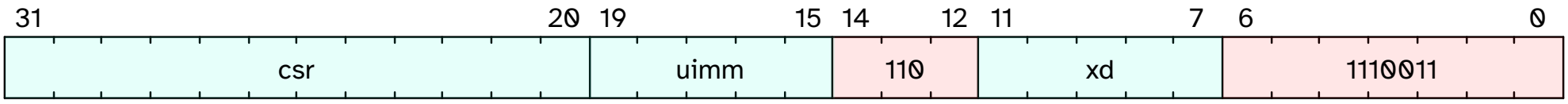
B.39. csrrsi

Atomic Read and Set Bits in CSR with Immediate

This instruction is defined by:

- Zicsr, version >= Zicsr@2.0.0

B.39.1. Encoding



B.39.2. 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 (uimm[4:0]) field encoded in the `rs1` field instead of a value from an integer register. For CSRRSI, if the `uimm[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 `rd` and `rs1` fields.

B.39.3. Access

M	S	U
Always	Always	Always

B.39.4. Decode Variables

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

B.39.5. IDL Operation

```
Boolean will_write = uimm != 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 = uimm;
  csr_sw_write(csr_handle, initial_csr_value | mask);
}
X[xd] = initial_csr_value;
```

B.39.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

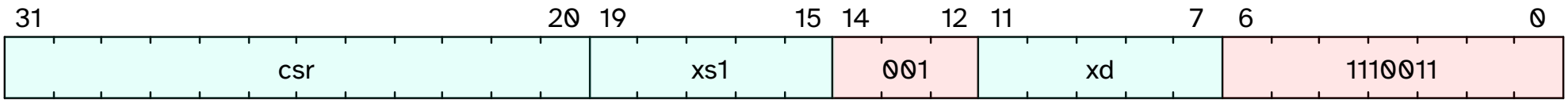
B.40. csrrw

Atomic Read/Write CSR

This instruction is defined by:

- Zicsr, version >= Zicsr@2.0.0

B.40.1. Encoding



B.40.2. 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.40.3. Access

M	S	U
Always	Always	Always

B.40.4. Decode Variables

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

B.40.5. 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.40.6. 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.40.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

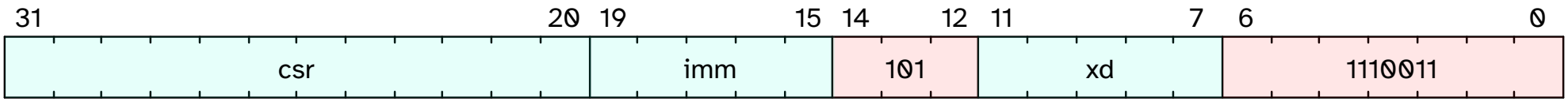
B.41. csrrwi

Atomic Read/Write CSR Immediate

This instruction is defined by:

- Zicsr, version >= Zicsr@2.0.0

B.41.1. Encoding



B.41.2. 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.41.3. Access

M	S	U
Always	Always	Always

B.41.4. Decode Variables

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

B.41.5. 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.41.6. 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.41.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

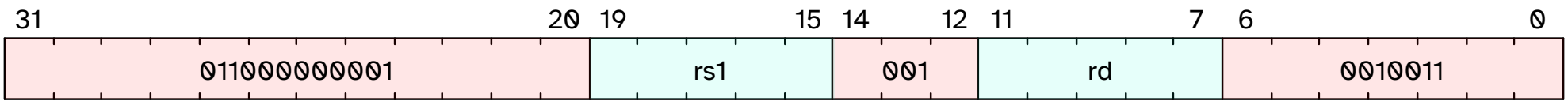
B.42. ctz

Count trailing zero bits

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.42.1. Encoding



B.42.2. Description

Counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit (i.e., XLEN-1). Accordingly, if the input is 0, the output is XLEN, and if the least-significant bit of the input is a 1, the output is 0.

B.42.3. Access

M	S	U
Always	Always	Always

B.42.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.42.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xlen() == 32) {
  X[rd] = lowest_set_bit(X[rs1][31:0]);
} else {
  X[rd] = lowest_set_bit(X[rs1]);
}
```

B.42.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : nat = 0;
  done : bool = false;
  foreach (i from 0 to (sizeof(xlen) - 1))
    if not(done) then if rs1_val[i] == bitzero
                      then result = result + 1
                      else done = true;
  X(rd) = to_bits(sizeof(xlen), result);
  RETIRE_SUCCESS
}
```

B.42.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

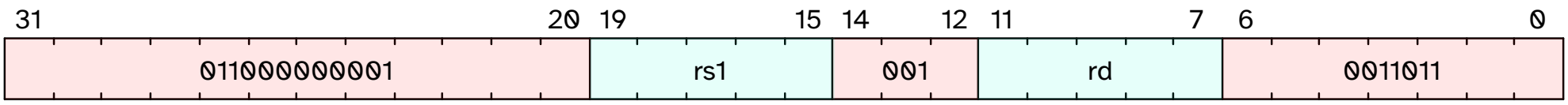
B.43. ctzw

Count trailing zero bits in word

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.43.1. Encoding



B.43.2. Description

Counts the number of 0's before the first 1, starting at the least-significant bit (i.e., 0) and progressing to the most-significant bit of the least-significant word (i.e., 31). Accordingly, if the least-significant word is 0, the output is 32, and if the least-significant bit of the input is a 1, the output is 0.

B.43.3. Access

M	S	U
Always	Always	Always

B.43.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.43.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = lowest_set_bit(X[rs1][31:0]);
```

B.43.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : nat = 0;
  done : bool = false;
  foreach (i from 0 to 31)
    if not(done) then if rs1_val[i] == bitzero
                      then result = result + 1
                      else done = true;
  X(rd) = to_bits(sizeof(xlen), result);
  RETIRE_SUCCESS
}
```

B.43.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

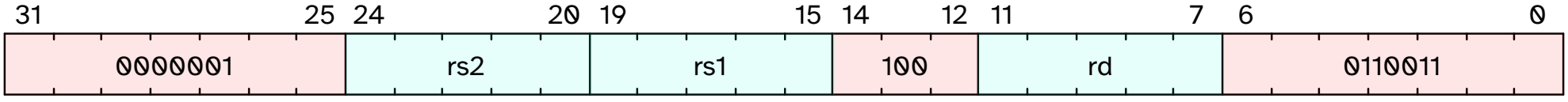
B.44. div

Signed division

This instruction is defined by:

- M, version >= M@2.0.0

B.44.1. Encoding



B.44.2. Description

Divide rs1 by rs2, and store the result in rd. The remainder is discarded.

Division by zero will put -1 into rd.

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

B.44.3. Access

M	S	U
Always	Always	Always

B.44.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.44.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
XReg signed_min = (xlen() == 32) ? $signed({1'b1, {31{1'b0}}}) : {1'b1, {63{1'b0}}};
if (src2 == 0) {
  X[rd] = {MXLEN{1'b1}};
} else if ((src1 == signed_min) && (src2 == {MXLEN{1'b1}})) {
  X[rd] = signed_min;
} else {
  X[rd] = $signed(src1) / $signed(src2);
}
```

B.44.6. 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.44.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

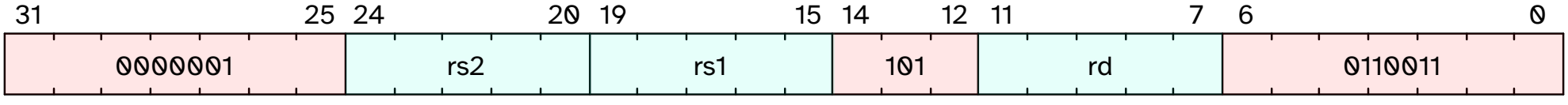
B.45. divu

Unsigned division

This instruction is defined by:

- M, version >= M@2.0.0

B.45.1. Encoding



B.45.2. Description

Divide unsigned values in rs1 by rs2, and store the result in rd.

The remainder is discarded.

If the value in rs2 is zero, rd gets the largest unsigned value.

B.45.3. Access

M	S	U
Always	Always	Always

B.45.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.45.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = {MXLEN{1'b1}};
} else {
  X[rd] = src1 / src2;
}
```

B.45.6. 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.45.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

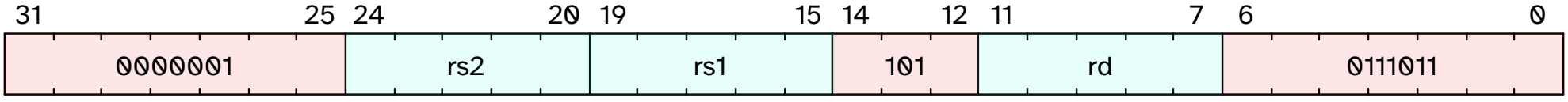
B.46. divuw

Unsigned 32-bit division

This instruction is defined by:

- M, version >= M@2.0.0

B.46.1. Encoding



B.46.2. Description

Divide the unsigned 32-bit values in rs1 and rs2, and store the sign-extended result in rd.

The remainder is discarded.

If the value in rs2 is zero, rd is written with all 1s.

B.46.3. Access

M	S	U
Always	Always	Always

B.46.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.46.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  X[rd] = {64{1'b1}};
} else {
  Bits<32> result = src1 / src2;
  Bits<1> sign_bit = result[31];
  X[rd] = {{32{sign_bit}}, result};
}
```

B.46.6. Sail Operation

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

B.46.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

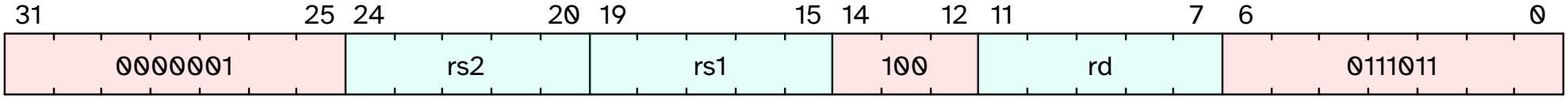
B.47. divw

Signed 32-bit division

This instruction is defined by:

- M, version >= M@2.0.0

B.47.1. Encoding



B.47.2. Description

Divide the lower 32-bits of register rs1 by the lower 32-bits of register rs2, and store the sign-extended result in rd.

The remainder is discarded.

Division by zero will put -1 into rd.

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

B.47.3. Access

M	S	U
Always	Always	Always

B.47.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.47.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  X[rd] = {MXLEN{1'b1}};
} else if ((src1 == {33'b1, 31'b0}) && (src2 == 32'b1)) {
  X[rd] = {33'b1, 31'b0};
} else {
  Bits<32> result = $signed(src1) / $signed(src2);
  Bits<1> sign_bit = result[31];
  X[rd] = {{32{sign_bit}}, result};
}
```

B.47.6. Sail Operation

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

```
    }
}
```

B.47.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

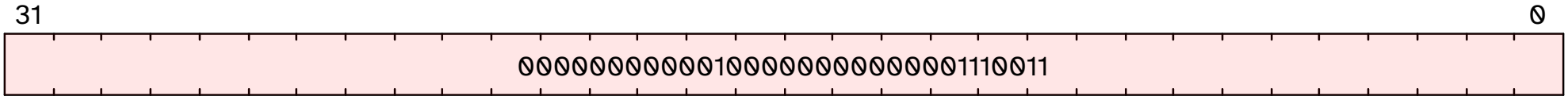
B.48. ebreak

Breakpoint exception

This instruction is defined by:

- I, version >= I@2.1.0

B.48.1. Encoding



B.48.2. 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.48.3. Access

M	S	U
Always	Always	Always

B.48.4. Decode Variables

```
```

B.48.5. IDL Operation

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

B.48.6. Sail Operation

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

B.48.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

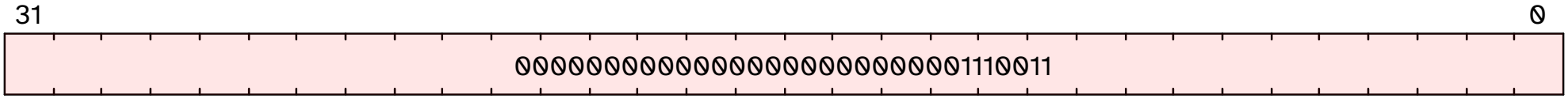
B.49. ecall

Environment call

This instruction is defined by:

- I, version >= I@2.1.0

B.49.1. Encoding



B.49.2. 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.49.3. Access

M	S	U
Always	Always	Always

B.49.4. Decode Variables

B.49.5. 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.49.6. 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.49.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

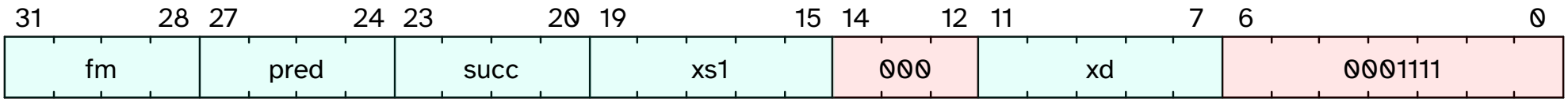
B.50. fence

Memory oxdering fence

This instruction is defined by:

- I, version >= I@2.1.0

B.50.1. Encoding



B.50.2. Description

Oxders memory operations.

The fence instruction is used to oxder 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 oxdered 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 oxdering; 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 oxders 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 unoxdered 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

Table 8. Effective PR/PW/SR/SW in U-mode

<%- if ext?(:H) -%> .Effective PR/PW/SR/SW in VS-mode and VU-mode

<%- end -%>

B.50.3. Access

M	S	U
Always	Always	Always

B.50.4. 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.50.5. IDL Operation

```
Boolean is_pause;
if (implemented?(ExtensionName::Zihintpause)) {
  if ((pred == 1) && (succ == 0) && (xd == 0) && (xs1 == 0)) {
    is_pause = true;
  }
}
Boolean pred_i = pred[3] == 1;
Boolean pred_o = pred[2] == 1;
Boolean pred_r = pred[1] == 1;
Boolean pred_w = pred[0] == 1;
Boolean succ_i = succ[3] == 1;
Boolean succ_o = succ[2] == 1;
Boolean succ_r = succ[1] == 1;
Boolean succ_w = succ[0] == 1;
if (is_pause) {
  pause();
} else {
  if (mode() == PrivilegeMode::S) {
    if (menvcfg.FIOM == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  } else if (mode() == PrivilegeMode::U) {
    if ((menvcfg.FIOM | senvcfg.FIOM) == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  } else if (mode() == PrivilegeMode::VS || mode() == PrivilegeMode::VU) {
    if ((menvcfg.FIOM | henvcfg.FIOM) == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  }
  fence(pred_i, pred_o, pred_r, pred_w, succ_i, succ_o, succ_r, succ_w);
}
```

B.50.6. 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.50.7. Exceptions

This instruction does not generate synchronous exceptions.

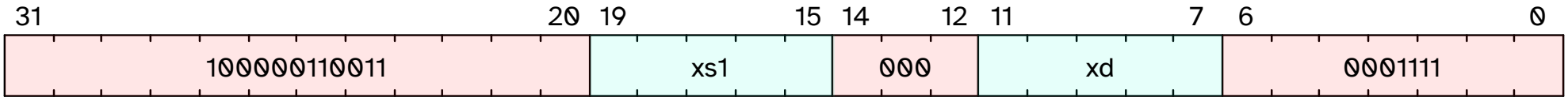
B.51. fence.tso

Memory ordering fence, total store ordering

This instruction is defined by:

- I, version >= I@2.1.0

B.51.1. Encoding



B.51.2. 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.51.3. Access

M	S	U
Always	Always	Always

B.51.4. Decode Variables

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

B.51.5. IDL Operation

```
fence_tso();
```

B.51.6. 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.51.7. Exceptions

This instruction does not generate synchronous exceptions.

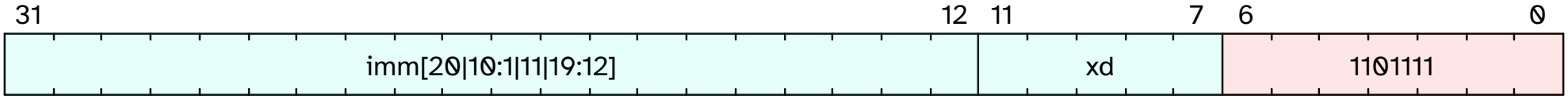
B.52. jal

Jump and link

This instruction is defined by:

- I, version >= I@2.1.0

B.52.1. Encoding



B.52.2. Description

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

B.52.3. Access

M	S	U
Always	Always	Always

B.52.4. 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.52.5. IDL Operation

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

B.52.6. 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.52.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

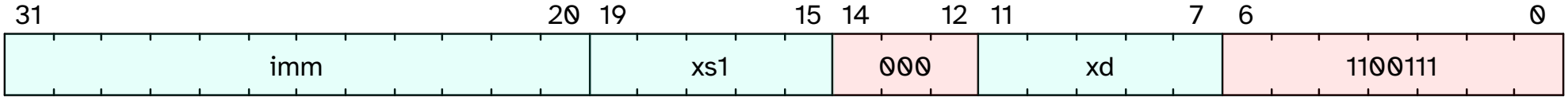
B.53. jalr

Jump and link register

This instruction is defined by:

- I, version >= I@2.1.0

B.53.1. Encoding



B.53.2. 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.53.3. Access

M	S	U
Always	Always	Always

B.53.4. Decode Variables

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

B.53.5. IDL Operation

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

B.53.6. 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.53.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

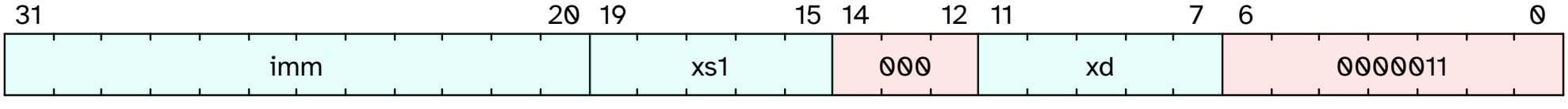
B.54. lb

Load byte

This instruction is defined by:

- I, version >= I@2.1.0

B.54.1. Encoding



B.54.2. 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.54.3. Access

M	S	U
Always	Always	Always

B.54.4. Decode Variables

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

B.54.5. IDL Operation

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

B.54.6. 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.54.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

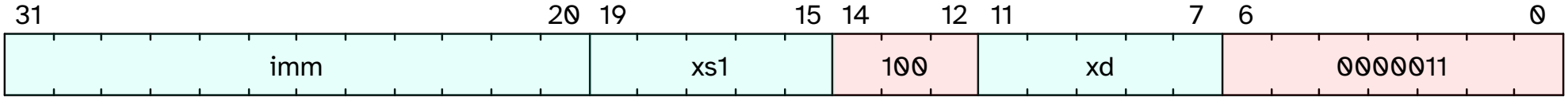
B.55. lbu

Load byte unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.55.1. Encoding



B.55.2. 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.55.3. Access

M	S	U
Always	Always	Always

B.55.4. Decode Variables

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

B.55.5. IDL Operation

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

B.55.6. 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.55.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

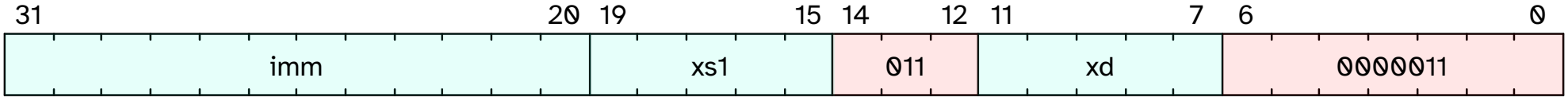
B.56. ld

Load doubleword

This instruction is defined by:

- I, version >= I@2.1.0

B.56.1. Encoding



B.56.2. Description

Load 64 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset.

B.56.3. Access

M	S	U
Always	Always	Always

B.56.4. Decode Variables

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

B.56.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
X[xd] = read_memory<64>(virtual_address, $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, 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.56.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

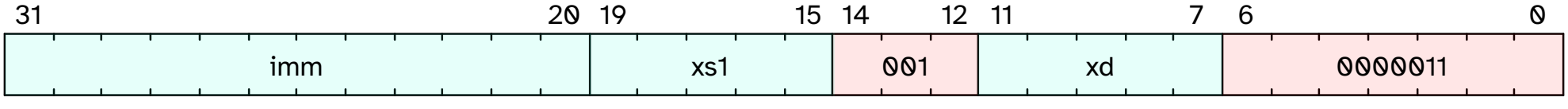
B.57. lh

Load halfword

This instruction is defined by:

- I, version >= I@2.1.0

B.57.1. Encoding



B.57.2. 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.57.3. Access

M	S	U
Always	Always	Always

B.57.4. Decode Variables

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

B.57.5. IDL Operation

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

B.57.6. 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.57.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

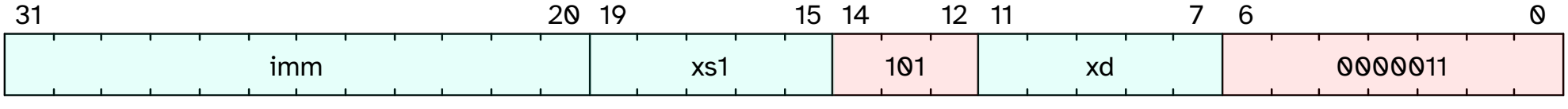
B.58. lhu

Load halfword unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.58.1. Encoding



B.58.2. 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.58.3. Access

M	S	U
Always	Always	Always

B.58.4. Decode Variables

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

B.58.5. IDL Operation

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

B.58.6. 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.58.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

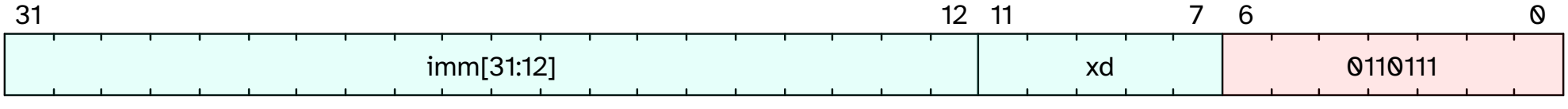
B.59. lui

Load upper immediate

This instruction is defined by:

- I, version >= I@2.1.0

B.59.1. Encoding



B.59.2. Description

Load the zero-extended imm into xd.

B.59.3. Access

M	S	U
Always	Always	Always

B.59.4. Decode Variables

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

B.59.5. IDL Operation

```
X[xd] = imm;
```

B.59.6. 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.59.7. Exceptions

This instruction does not generate synchronous exceptions.

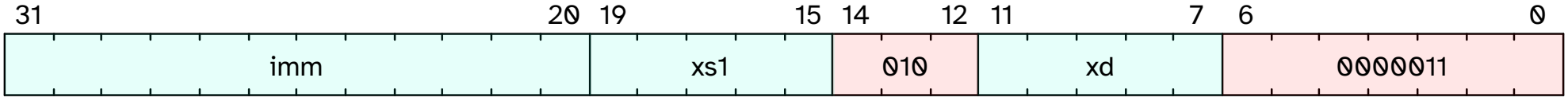
B.60. lw

Load word

This instruction is defined by:

- I, version >= I@2.1.0

B.60.1. Encoding



B.60.2. 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.60.3. Access

M	S	U
Always	Always	Always

B.60.4. Decode Variables

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

B.60.5. IDL Operation

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

B.60.6. 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.60.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

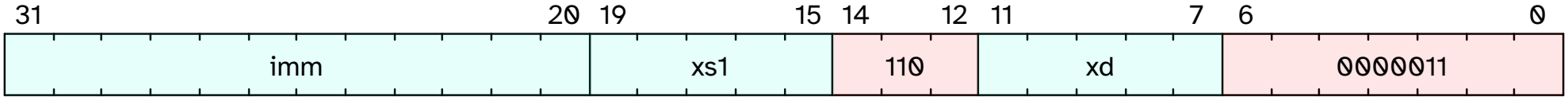
B.61. lwu

Load word unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.61.1. Encoding



B.61.2. Description

Load 64 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Zero extend the result.

B.61.3. Access

M	S	U
Always	Always	Always

B.61.4. Decode Variables

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

B.61.5. IDL Operation

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

B.61.6. 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.61.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

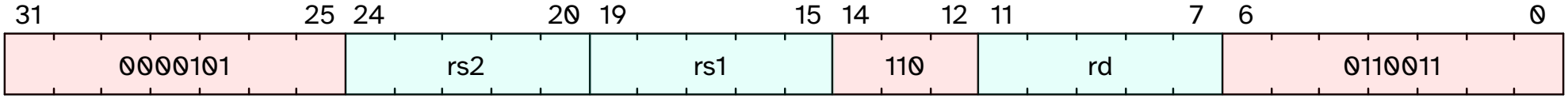
B.62. max

Maximum

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.62.1. Encoding



B.62.2. Description

Returns the larger of two signed integers.

Software Hint

Calculating the absolute value of a signed integer can be performed using the following sequence: `neg rD,rS` *followed by `max rD,rS,rD. When using this common sequence, it is suggested that they are scheduled with no intervening instructions so that implementations that are so optimized can fuse them together.*

B.62.3. Access

M	S	U
Always	Always	Always

B.62.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.62.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = ($signed(X[rs1]) > $signed(X[rs2])) ? X[rs1] : X[rs2];
```

B.62.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.62.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

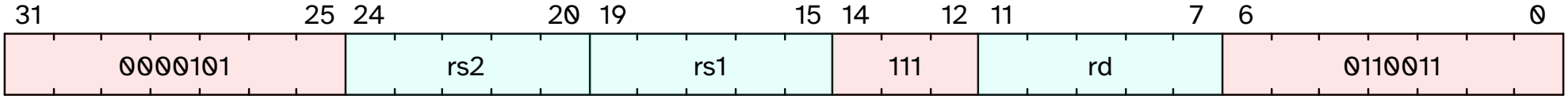
B.63. maxu

Unsigned maximum

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.63.1. Encoding



B.63.2. Description

Returns the larger of two unsigned integers.

B.63.3. Access

M	S	U
Always	Always	Always

B.63.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.63.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = (X[rs1] > X[rs2]) ? X[rs1] : X[rs2];
```

B.63.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.63.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

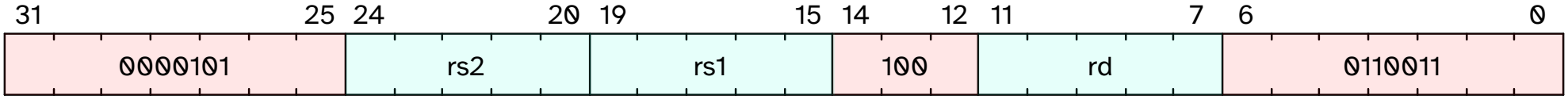
B.64. min

Minimum

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.64.1. Encoding



B.64.2. Description

Returns the smaller of two signed integers.

B.64.3. Access

M	S	U
Always	Always	Always

B.64.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.64.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = ($signed(X[rs1]) < $signed(X[rs2])) ? X[rs1] : X[rs2];
```

B.64.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.64.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

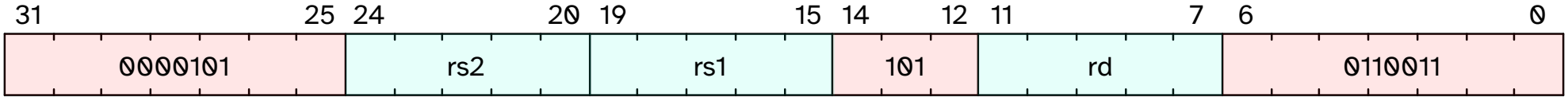
B.65. minu

Unsigned minimum

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.65.1. Encoding



B.65.2. Description

Returns the smaller of two unsigned integers.

B.65.3. Access

M	S	U
Always	Always	Always

B.65.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.65.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = (X[rs1] < X[rs2]) ? X[rs1] : X[rs2];
```

B.65.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.65.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.66. mret

Machine Exception Return

This instruction is defined by:

- Sm, version >= Sm@1.11.0

B.66.1. Encoding

B.66.2. Description

Returns from an exception in M-mode.

B.66.3. Access

M	S	U
Always	Never	Never

B.66.4. Decode Variables

```
```

B.66.5. IDL Operation

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

B.66.6. 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.66.7. Exceptions

This instruction does not generate synchronous exceptions.

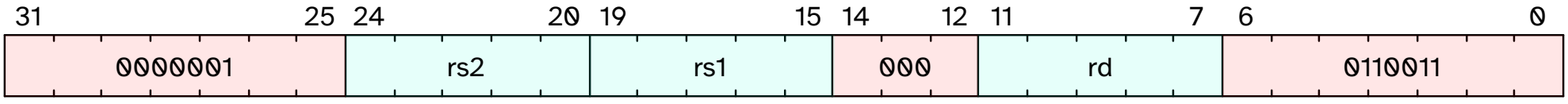
B.67. mul

Signed multiply

This instruction is defined by:

- anyOf:
  - M, version >= M@2.0.0
  - Zmmul, version >= Zmmul@1.0.0

B.67.1. Encoding



B.67.2. Description

MUL performs an XLEN-bitxXLEN-bit multiplication of `rs1` by `rs2` 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] rdh, rs1, rs2; MUL rdl, rs1, rs2 (source register specifiers must be in same order and rdh cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

B.67.3. Access

M	S	U
Always	Always	Always

B.67.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.67.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
X[rd] = (src1 * src2)[MXLEN - 1:0];
```

B.67.6. 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.67.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

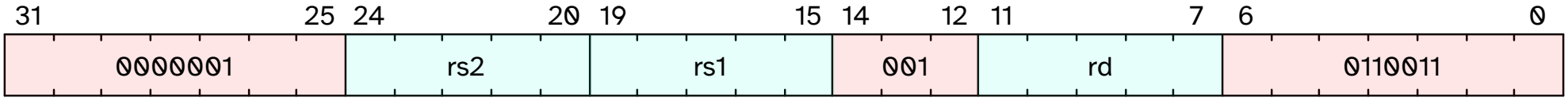
B.68. mulh

Signed multiply high

This instruction is defined by:

- anyOf:
  - M, version >= M@2.0.0
  - Zmmul, version >= Zmmul@1.0.0

B.68.1. Encoding



B.68.2. Description

Multiply the signed values in rs1 to rs2, and store the upper half of the result in rd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
  mulh rdh, rs1, rs2
  mul  rdl, rs1, rs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.68.3. Access

M	S	U
Always	Always	Always

B.68.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.68.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<1> rs1_sign_bit = X[rs1][xlen() - 1];
Bits<MXLEN `* 2> src1 = {{xlen(){rs1_sign_bit}}, X[rs1]};
Bits<1> rs2_sign_bit = X[rs2][xlen() - 1];
Bits<MXLEN `* 2> src2 = {{xlen(){rs2_sign_bit}}, X[rs2]};
X[rd] = (src1 * src2)[(xlen() * 8'd2) - 1:xlen()];
```

B.68.6. 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.68.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

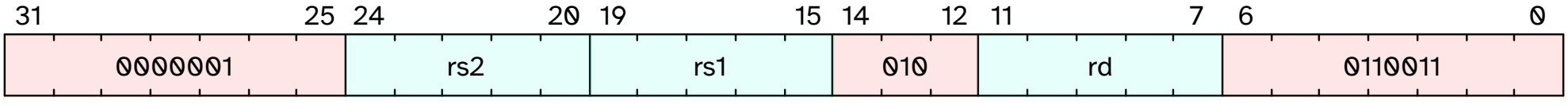
B.69. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

- anyOf:
  - M, version >= M@2.0.0
  - Zmmul, version >= Zmmul@1.0.0

B.69.1. Encoding



B.69.2. Description

Multiply the signed value in rs1 by the unsigned value in rs2, and store the upper half of the result in rd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
  mulhsu rdh, rs1, rs2
  mul    rdl, rs1, rs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.69.3. Access

M	S	U
Always	Always	Always

B.69.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.69.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<1> rs1_sign_bit = X[rs1][MXLEN - 1];
Bits<MXLEN * 8'd2> src1 = {{MXLEN{rs1_sign_bit}}, X[rs1]};
Bits<MXLEN * 8'd2> src2 = {{MXLEN{1'b0}}, X[rs2]};
X[rd] = (src1 * src2)[(MXLEN * 8'd2) - 1:MXLEN];
```

B.69.6. 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.69.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

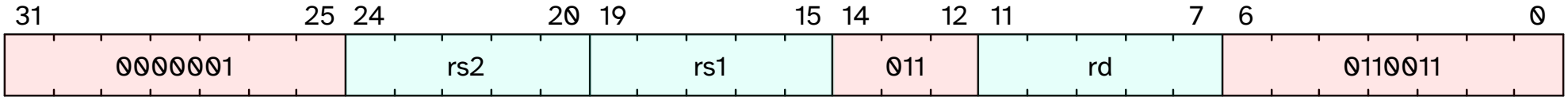
B.70. mulhu

Unsigned multiply high

This instruction is defined by:

- anyOf:
  - M, version >= M@2.0.0
  - Zmmul, version >= Zmmul@1.0.0

B.70.1. Encoding



B.70.2. Description

Multiply the unsigned values in rs1 to rs2, and store the upper half of the result in rd. The lower half is thrown away.

If both the upper and lower halves are needed, it suggested to use the sequence:

```
  mulhu rdh, rs1, rs2
  mul   rdl, rs1, rs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

B.70.3. Access

M	S	U
Always	Always	Always

B.70.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.70.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<MXLEN * 8'd2> src1 = {{MXLEN{1'b0}}, X[rs1]};
Bits<MXLEN * 8'd2> src2 = {{MXLEN{1'b0}}, X[rs2]};
X[rd] = (src1 * src2)[(MXLEN * 8'd2) - 1:MXLEN];
```

B.70.6. 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.70.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

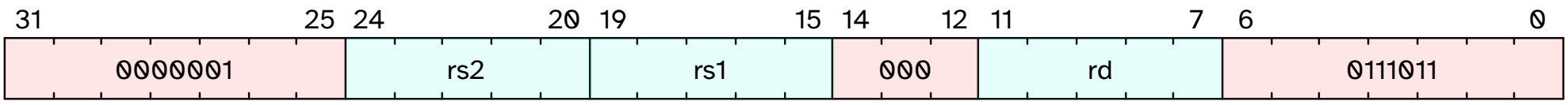
B.71. mulw

Signed 32-bit multiply

This instruction is defined by:

- anyOf:
  - M, version >= M@2.0.0
  - Zmmul, version >= Zmmul@1.0.0

B.71.1. Encoding



B.71.2. Description

Multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register.

Any overflow is thrown away.

In RV64, MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear. If the arguments are not known to be sign- or zero-extended, an alternative is to shift both arguments left by 32 bits, then use MULH[[S]U].

B.71.3. Access

M	S	U
Always	Always	Always

B.71.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.71.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
Bits<32> result = src1 * src2;
Bits<1> sign_bit = result[31];
X[rd] = {{32{sign_bit}}, result};
```

B.71.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = signed(rs1_val);
    let rs2_int : int = signed(rs2_val);
    /* to_bits requires expansion to 64 bits followed by truncation */
    let result32 = to_bits(64, rs1_int * rs2_int)[31..0];
    let result : xlenbits = sign_extend(result32);
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.71.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

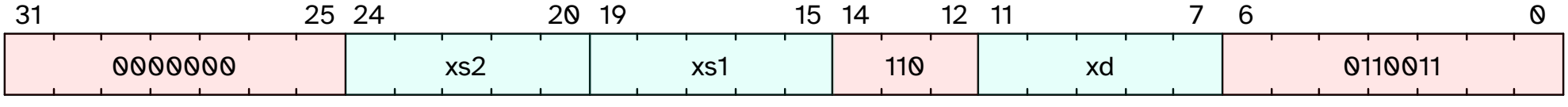
B.72. or

Or

This instruction is defined by:

- I, version >= I@2.1.0

B.72.1. Encoding



B.72.2. Description

Or xs1 with xs2, and store the result in xd

B.72.3. Access

M	S	U
Always	Always	Always

B.72.4. Decode Variables

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

B.72.5. IDL Operation

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

B.72.6. 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.72.7. Exceptions

This instruction does not generate synchronous exceptions.

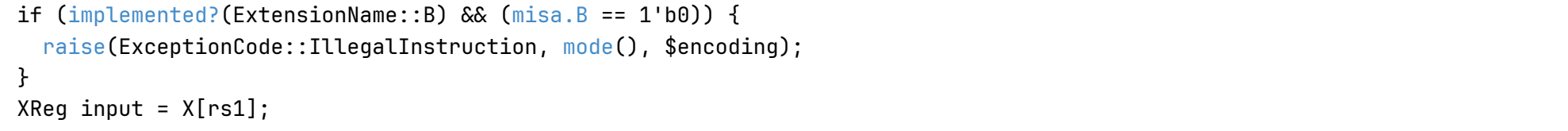
B.73. orc.b

Bitware OR-combine, byte granule

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.73.1. Encoding



B.73.2. Description

Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result rd to all zeros if no bit within the respective byte of rs is set, or to all ones if any bit within the respective byte of rs is set.

B.73.3. Access

M	S	U
Always	Always	Always

B.73.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.73.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg input = X[rs1];
XReg output = 0;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
  output[(i + 7):i] = (input[(i + 7):i] == 0) ? 8'd0 : ~8'd0;
}
X[rd] = output;
```

B.73.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : xlenbits = zeros();
  foreach (i from 0 to (sizeof(xlen) - 8) by 8)
    result[(i + 7) .. i] = if rs1_val[(i + 7) .. i] == zeros()
                           then 0x00
                           else 0xFF;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.73.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.74. ori

Or immediate

This instruction is defined by:

- I, version >= I@2.1.0

B.74.1. Encoding

B.74.2. Description

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

B.74.3. Access

M	S	U
Always	Always	Always

B.74.4. Decode Variables

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

B.74.5. IDL Operation

```
if (implemented?(ExtensionName::Zicbop)) {
  if (xd == 0) {
    if (imm[4:0] == 0) {
      Bits<12> offset = {imm[11:5], xd};
      prefetch_instruction(offset);
    } else if (imm[4:0] == 1) {
      Bits<12> offset = {imm[11:5], xd};
      prefetch_read(offset);
    } else if (imm[4:0] == 3) {
      Bits<12> offset = {imm[11:5], xd};
      prefetch_write(offset);
    }
  }
}
X[xd] = X[xs1] | $signed(imm);
```

B.74.6. 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.74.7. Exceptions

This instruction does not generate synchronous exceptions.

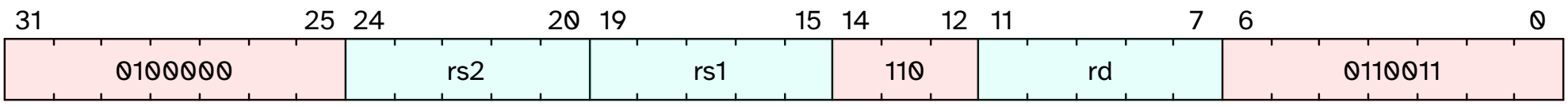
B.75. orn

OR with inverted operand

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.75.1. Encoding



B.75.2. Description

Performs the bitwise logical OR operation between rs1 and the bitwise inversion of rs2.

B.75.3. Access

M	S	U
Always	Always	Always

B.75.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.75.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs1] | ~X[rs2];
```

B.75.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.75.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

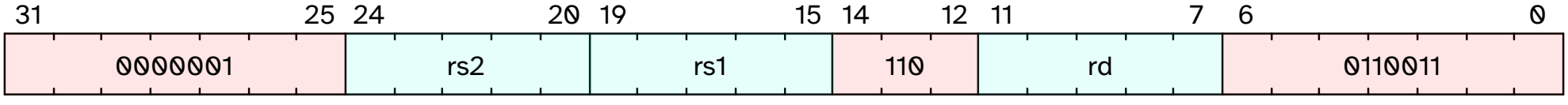
B.76. rem

Signed remainder

This instruction is defined by:

- M, version >= M@2.0.0

B.76.1. Encoding



B.76.2. Description

Calculate the remainder of signed division of rs1 by rs2, and store the result in rd.

If the value in register rs2 is zero, write the value in rs1 into rd;

If the result of the division overflows, write zero into rd;

B.76.3. Access

M	S	U
Always	Always	Always

B.76.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.76.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = src1;
} else if ((src1 == {1'b1, {MXLEN - 1{1'b0}}}) && (src2 == {MXLEN{1'b1}})) {
  X[rd] = 0;
} else {
  X[rd] = $signed(src1) % $signed(src2);
}
```

B.76.6. 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.76.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

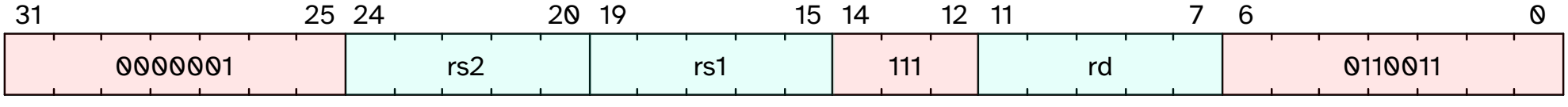
B.77. remu

Unsigned remainder

This instruction is defined by:

- M, version >= M@2.0.0

B.77.1. Encoding



B.77.2. Description

Calculate the remainder of unsigned division of rs1 by rs2, and store the result in rd.

B.77.3. Access

M	S	U
Always	Always	Always

B.77.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.77.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg src1 = X[rs1];
XReg src2 = X[rs2];
if (src2 == 0) {
  X[rd] = src1;
} else {
  X[rd] = src1 % src2;
}
```

B.77.6. 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.77.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

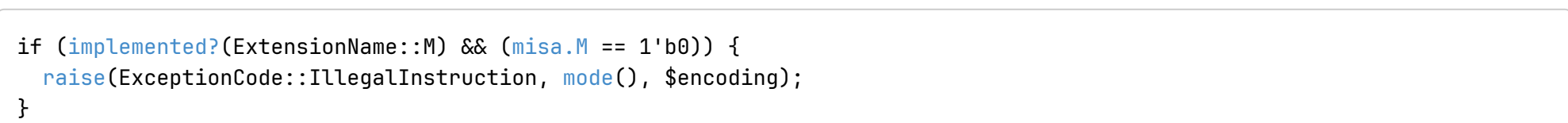
B.78. remuw

Unsigned 32-bit remainder

This instruction is defined by:

- M, version >= M@2.0.0

B.78.1. Encoding



B.78.2. Description

Calculate the remainder of unsigned division of the 32-bit values in rs1 by rs2, and store the sign-extended result in rd.

If the value in rs2 is zero, rd gets the sign-extended value in rs1.

B.78.3. Access

M	S	U
Always	Always	Always

B.78.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.78.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  Bits<1> sign_bit = src1[31];
  X[rd] = {{32{sign_bit}}, src1};
} else {
  Bits<32> result = src1 % src2;
  Bits<1> sign_bit = result[31];
  X[rd] = {{32{sign_bit}}, result};
}
```

B.78.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let r : int = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);
    /* signed overflow case returns zero naturally as required due to -1 divisor */
    X(rd) = sign_extend(to_bits(32, r));
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

B.78.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

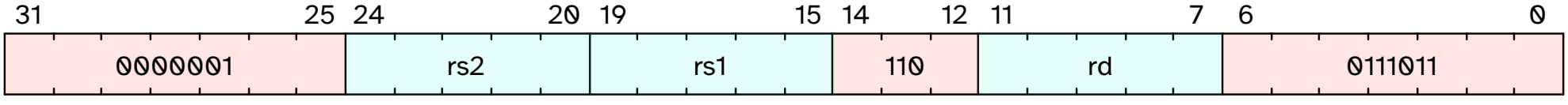
B.79. remw

Signed 32-bit remainder

This instruction is defined by:

- M, version >= M@2.0.0

B.79.1. Encoding



B.79.2. Description

Calculate the remainder of signed division of the 32-bit values rs1 by rs2, and store the sign-extended result in rd.

If the value in register rs2 is zero, write the sign-extended 32-bit value in rs1 into rd;

If the result of the division overflows, write zero into rd;

B.79.3. Access

M	S	U
Always	Always	Always

B.79.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.79.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (misa.M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<32> src1 = X[rs1][31:0];
Bits<32> src2 = X[rs2][31:0];
if (src2 == 0) {
  Bits<1> sign_bit = src1[31];
  X[rd] = {{32{sign_bit}}, src1};
} else if ((src1 == {33'b1, 31'b0}) && (src2 == 32'b1)) {
  X[rd] = 0;
} else {
  Bits<32> result = $signed(src1) % $signed(src2);
  Bits<1> sign_bit = result[31];
  X[rd] = {{32{sign_bit}}, result};
}
```

B.79.6. Sail Operation

```
{
  if extension("M") then {
    let rs1_val = X(rs1)[31..0];
    let rs2_val = X(rs2)[31..0];
    let rs1_int : int = if s then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if s then signed(rs2_val) else unsigned(rs2_val);
    let r : int = if rs2_int == 0 then rs1_int else rem_round_zero(rs1_int, rs2_int);
    /* signed overflow case returns zero naturally as required due to -1 divisor */
    X(rd) = sign_extend(to_bits(32, r));
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
```

```
}
```

B.79.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.80. rev8

Byte-reverse register (RV64 encoding)

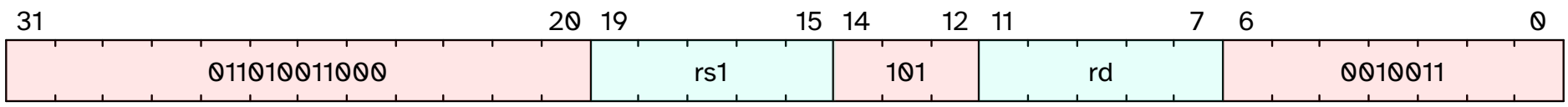
This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

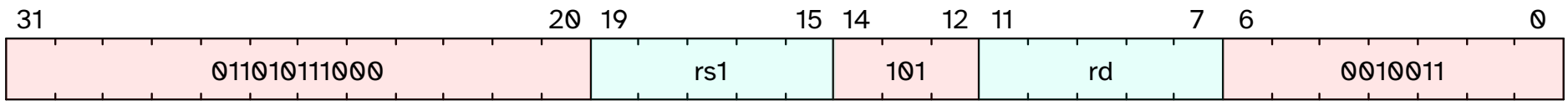
B.80.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.80.2. Description

Reverses the order of the bytes in rs1.

The rev8 mnemonic corresponds to different instruction encodings in RV32 and RV64.

The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a `rev8 rd,rs` followed by a `srai rd,rd,K`, where K is XLEN-32 and XLEN-16, respectively.

B.80.3. Access

M	S	U
Always	Always	Always

B.80.4. Decode Variables

RV32

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.80.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg input = X[rs1];
XReg output = 0;
XReg j = xlen() - 1;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
  output[(i + 7):i] = input[j:(j - 7)];
  j = j - 8;
}
X[rd] = output;
```

B.80.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  result : xlenbits = zeros();
  foreach (i from 0 to (sizeof(xlen) - 8) by 8)
    result[(i + 7) .. i] = rs1_val[(sizeof(xlen) - i - 1) .. (sizeof(xlen) - i - 8)];
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.80.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

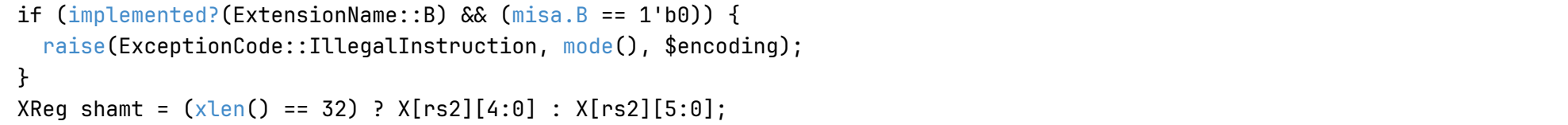
B.81. rol

Rotate left (Register)

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.81.1. Encoding



B.81.2. Description

Performs a rotate left of rs1 by the amount in least-significant `log2(XLEN)` bits of rs2.

B.81.3. Access

M	S	U
Always	Always	Always

B.81.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.81.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg shamt = (xlen() == 32) ? X[rs2][4:0] : X[rs2][5:0];
X[rd] = (X[rs1] << shamt) | (X[rs1] >> (xlen() - shamt));
```

B.81.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.81.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

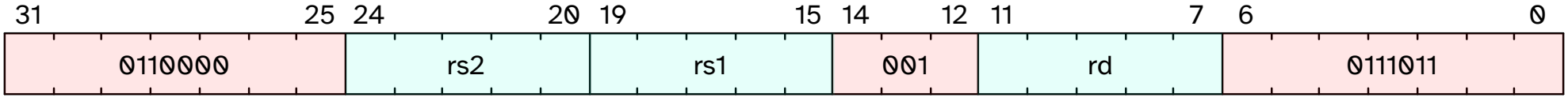
B.82. rolw

Rotate left word (Register)

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.82.1. Encoding



B.82.2. Description

Performs a rotate left of the least-significant word of rs1 by the amount in least-significant 5 bits of rs2. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

B.82.3. Access

M	S	U
Always	Always	Always

B.82.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.82.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg rs1_word = X[rs1][31:0];
XReg shamt = X[rs2][4:0];
XReg unextended_result = (rs1_word << shamt) | (rs1_word >> (32 - shamt));
X[rd] = {{32{unextended_result[31]}}, unextended_result};
```

B.82.6. Sail Operation

```
{
  let rs1_val = (X(rs1))[31..0];
  let shamt = (X(rs2))[4..0];
  let result : bits(32) = match op {
    RISCV_ROLW => rs1_val <<< shamt,
    RISCV_RORW => rs1_val >>> shamt
  };
  X(rd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.82.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

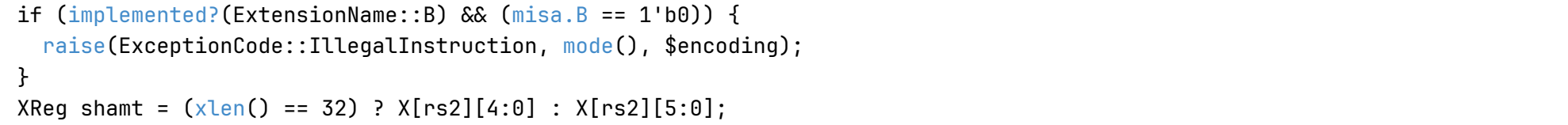
B.83. ror

Rotate right (Register)

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.83.1. Encoding



B.83.2. Description

Performs a rotate right of rs1 by the amount in least-significant `log2(XLEN)` bits of rs2.

B.83.3. Access

M	S	U
Always	Always	Always

B.83.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.83.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg shamt = (xlen() == 32) ? X[rs2][4:0] : X[rs2][5:0];
X[rd] = (X[rs1] >> shamt) | (X[rs1] << (xlen() - shamt));
```

B.83.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.83.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.84. rori

Rotate right (Immediate)

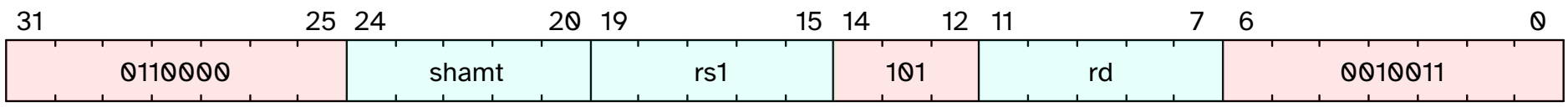
This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

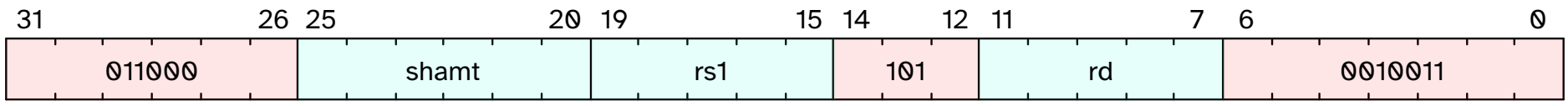
B.84.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.84.2. Description

Performs a rotate right of rs1 by the amount in the least-significant log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

B.84.3. Access

M	S	U
Always	Always	Always

B.84.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

RV64

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.84.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg shamt = (xlen() == 32) ? shamt[4:0] : shamt[5:0];
X[rd] = (X[rs1] >> shamt) | (X[rs1] << (xlen() - shamt));
```

B.84.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let result : xlenbits = if sizeof(xlen) == 32
                          then rs1_val >>> shamt[4..0]
                          else rs1_val >>> shamt;
```

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

B.84.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

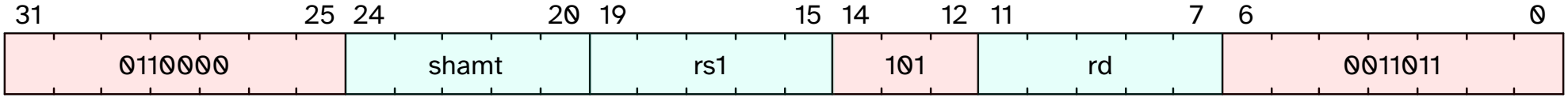
B.85. roriw

Rotate right word (Immediate)

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.85.1. Encoding



B.85.2. Description

Performs a rotate right on the least-significant word of rs1 by the amount in the least-significant log2(XLEN) bits of shamt. The resulting word value is sign-extended by copying bit 31 to all of the more-significant bits.

B.85.3. Access

M	S	U
Always	Always	Always

B.85.4. Decode Variables

```
Bits<5> shamt = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.85.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg rs1_word = X[rs1][31:0];
XReg unextended_result = (X[rs1] >> shamt) | (X[rs1] << (32 - shamt));
X[rd] = {{32{unextended_result[31]}}, unextended_result};
```

B.85.6. Sail Operation

```
{
  let rs1_val = (X(rs1))[31..0];
  let result : xlenbits = sign_extend(rs1_val >>> shamt);
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.85.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

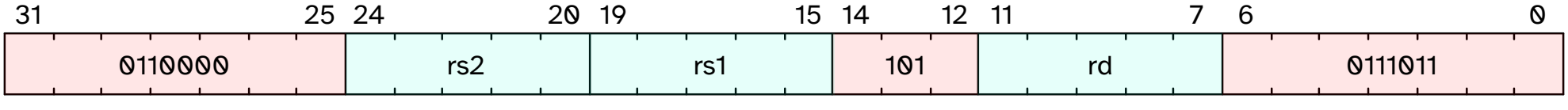
B.86. rorw

Rotate right word (Register)

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.86.1. Encoding



B.86.2. Description

Performs a rotate right on the least-significant word of rs1 by the amount in least-significant 5 bits of rs2. The resultant word is sign-extended by copying bit 31 to all of the more-significant bits.

B.86.3. Access

M	S	U
Always	Always	Always

B.86.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.86.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg rs1_word = X[rs1][31:0];
XReg shamt = X[rs2][4:0];
XReg unextended_result = (X[rs1] >> shamt) | (X[rs1] << (32 - shamt));
X[rd] = {{32{unextended_result[31]}}, unextended_result};
```

B.86.6. Sail Operation

```
{
  let rs1_val = (X(rs1))[31..0];
  let shamt = (X(rs2))[4..0];
  let result : bits(32) = match op {
    RISCV_ROLW => rs1_val <<< shamt,
    RISCV_RORW => rs1_val >>> shamt
  };
  X(rd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.86.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

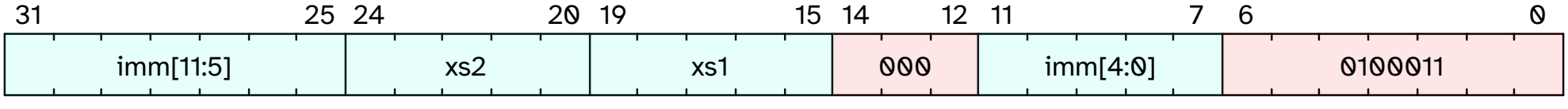
B.87. sb

Store byte

This instruction is defined by:

- I, version >= I@2.1.0

B.87.1. Encoding



B.87.2. Description

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

B.87.3. Access

M	S	U
Always	Always	Always

B.87.4. Decode Variables

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

B.87.5. IDL Operation

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

B.87.6. 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.87.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

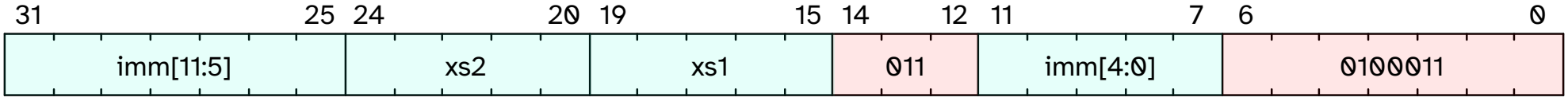
B.88. sd

Store doubleword

This instruction is defined by:

- I, version >= I@2.1.0

B.88.1. Encoding



B.88.2. Description

Store 64 bits of data from register `xs2` to an address formed by adding `xs1` to a signed offset.

B.88.3. Access

M	S	U
Always	Always	Always

B.88.4. Decode Variables

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

B.88.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<64>(virtual_address, X[xs2], $encoding);
```

B.88.6. 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.88.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

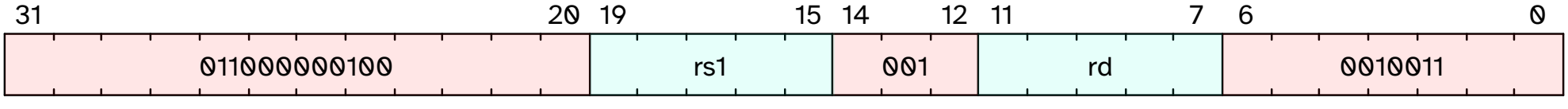
B.89. sext.b

Sign-extend byte

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.89.1. Encoding



B.89.2. Description

Sign-extends the least-significant byte in the source to XLEN by copying the most-significant bit in the byte (i.e., bit 7) to all of the more-significant bits.

B.89.3. Access

M	S	U
Always	Always	Always

B.89.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.89.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xlen() == 32) {
  X[rd] = {{24{X[rs1][7]}}, X[rs1][7:0]};
} else if (xlen() == 64) {
  X[rd] = {{56{X[rs1][7]}}, X[rs1][7:0]};
}
```

B.89.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let result : xlenbits = match op {
    RISCV_SEXTB => sign_extend(rs1_val[7..0]),
    RISCV_SEXTH => sign_extend(rs1_val[15..0]),
    RISCV_ZEXTH => zero_extend(rs1_val[15..0])
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.89.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

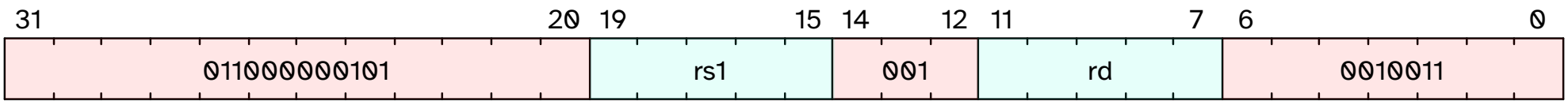
B.90. sext.h

Sign-extend halfword

This instruction is defined by:

- Zbb, version >= Zbb@1.0.0

B.90.1. Encoding



B.90.2. Description

Sign-extends the least-significant halfword in the source to XLEN by copying the most-significant bit in the halfword (i.e., bit 15) to all of the more-significant bits.

B.90.3. Access

M	S	U
Always	Always	Always

B.90.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.90.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xlen() == 32) {
  X[rd] = {{16{X[rs1][15]}}, X[rs1][15:0]};
} else if (xlen() == 64) {
  X[rd] = {{48{X[rs1][15]}}, X[rs1][15:0]};
}
```

B.90.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let result : xlenbits = match op {
    RISCV_SEXTB => sign_extend(rs1_val[7..0]),
    RISCV_SEXTH => sign_extend(rs1_val[15..0]),
    RISCV_ZEXTH => zero_extend(rs1_val[15..0])
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.90.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

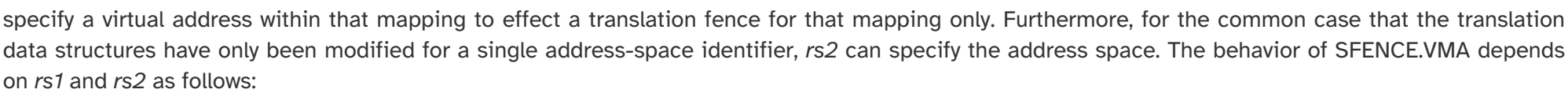
B.91. sfence.vma

Supervisor memory-management fence

This instruction is defined by:

- S, version >= S@1.11.0

B.91.1. Encoding



B.91.2. Description

The supervisor memory-management fence instruction `SFENCE.VMA` is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by *rs1* and *rs2*, as described below. SFENCE.VMA is also used to invalidate entries in the address-translation cache associated with a hart (see [sv32algorithm]). Further details on the behavior of this instruction are described in [virt-control] and [pmp-vmem].

The SFENCE.VMA is used to flush any local hardware caches related to address translation. It is specified as a fence rather than a TLB flush to provide cleaner semantics with respect to which instructions are affected by the flush operation and to support a wider variety of dynamic caching structures and memory-management schemes. SFENCE.VMA is also used by higher privilege levels to synchronize page table writes and the address translation hardware.

SFENCE.VMA orders only the local hart's implicit references to the memory-management data structures.

Consequently, other harts must be notified separately when the memory-management data structures have been modified. One approach is to use 1) a local data fence to ensure local writes are visible globally, then 2) an interprocessor interrupt to the other thread, then 3) a local SFENCE.VMA in the interrupt handler of the remote thread, and finally 4) signal back to originating thread that operation is complete. This is, of course, the RISC-V analog to a TLB shootdown.

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), *rs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, *rs2* can specify the address space. The behavior of SFENCE.VMA depends on *rs1* and *rs2* as follows:

- If *rs1*=`x0` and *rs2*=`x0`, the fence orders all reads and writes made to any level of the page tables, for all address spaces. The fence also invalidates all address-translation cache entries, for all address spaces.
- If *rs1*=`x0` and *rs2*≠`x0`, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register *rs2*. Accesses to *global* mappings (see [translation]) are not ordered. The fence also invalidates all address-translation cache entries matching the address space identified by integer register *rs2*, except for entries containing global mappings.
- If *rs1*≠`x0` and *rs2*=`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1*, for all address spaces.
- If *rs1*≠`x0` and *rs2*≠`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *rs1*, for the address space identified by integer register *rs2*. Accesses to global mappings are not ordered. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *rs1* and that match the address space identified by integer register *rs2*, except for entries containing global mappings.

If the value held in *rs1* is not a valid virtual address, then the SFENCE.VMA instruction has no effect. No exception is raised in this case.

When *rs2*≠`x0`, bits SXLEN-1:ASIDMAX of the value held in *rs2* are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLEN<ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN of the value held in *rs2*.

It is always legal to over-fence, e.g., by fencing only based on a subset of the bits in rs1 and/or rs2, and/or by simply treating all SFENCE.VMA instructions as having rs1=`x0` and/or rs2=`x0`. For example, simpler implementations can ignore the virtual address in rs1 and the ASID value in rs2 and always perform a global fence. The choice not to raise an exception when an invalid virtual address is held in rs1 facilitates this type of simplification.

An implicit read of the memory-management data structures may return any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. The ordering implied by SFENCE.VMA does not place implicit reads and writes to the memory-management data structures into the global memory order in a way that interacts cleanly with the standard RVWMO ordering rules. In particular, even though an SFENCE.VMA orders prior explicit accesses before subsequent implicit accesses, and those implicit accesses are ordered before their associated explicit accesses, SFENCE.VMA does not necessarily place prior explicit accesses before subsequent explicit accesses in the global memory order. These implicit loads also need not otherwise obey normal program order semantics with respect to prior loads or stores to the same address.

A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. In particular, if a leaf PTE is modified but a subsuming SFENCE.VMA is not

> *executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.*
>
> *In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with rs1=`x0`. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.*
>
> *Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.*
>
> ---
>
> *This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.*

Implementations must only perform implicit reads of the translation data structures pointed to by the current contents of the satp register or a subsequent valid (V=1) translation data structure entry, and must only raise exceptions for implicit accesses that are generated as a result of instruction execution, not those that are performed speculatively.

Changes to the sstatus fields SUM and MXR take effect immediately, without the need to execute an SFENCE.VMA instruction. Changing satp.MODE from Bare to other modes and vice versa also takes effect immediately, without the need to execute an SFENCE.VMA instruction. Likewise, changes to satp.ASID take effect immediately.

> *The following common situations typically require executing an SFENCE.VMA instruction:*
>
> - *When software recycles an ASID (i.e., reassociates it with a different page table), it should first change satp to point to the new page table using the recycled ASID, then execute SFENCE.VMA with rs1=`x0` and rs2 set to the recycled ASID. Alternatively, software can execute the same SFENCE.VMA instruction while a different ASID is loaded into satp, provided the next time satp is loaded with the recycled ASID, it is simultaneously loaded with the new page table.*
> - *If the implementation does not provide ASIDs, or software chooses to always use ASID `0`, then after every satp write, software should execute SFENCE.VMA with rs1=`x0`. In the common case that no global translations have been modified, rs2 should be set to a register other than `x0` but which contains the value zero, so that global translations are not flushed.*
> - *If software modifies a non-leaf PTE, it should execute SFENCE.VMA with rs1=`x0`. If any PTE along the traversal path had its G bit set, rs2 must be `x0`; otherwise, rs2 should be set to the ASID for which the translation is being modified.*
> - *If software modifies a leaf PTE, it should execute SFENCE.VMA with rs1 set to a virtual address within the page. If any PTE along the traversal path had its G bit set, rs2 must be `x0`; otherwise, rs2 should be set to the ASID for which the translation is being modified.*
> - *For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the SFENCE.VMA lazily. After modifying the PTE but before executing SFENCE.VMA, either the new or old permissions will be used. In the latter case, a page-fault exception might occur, at which point software should execute SFENCE.VMA in accordance with the previous bullet point.*

If a hart employs an address-translation cache, that cache must appear to be private to that hart. In particular, the meaning of an ASID is local to a hart; software may choose to use the same ASID to refer to different address spaces on different harts.

> *A future extension could redefine ASIDs to be global across the SEE, enabling such options as shared translation caches and hardware support for broadcast TLB shootdown. However, as OSes have evolved to significantly reduce the scope of TLB shootdowns using novel ASID-management techniques, we expect the local-ASID scheme to remain attractive for its simplicity and possibly better scalability.*

For implementations that make satp.MODE read-only zero (always Bare), attempts to execute an SFENCE.VMA instruction might raise an illegal-instruction exception.

B.91.3. Access

M	S	U
Always	Always	Never

B.91.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.91.5. IDL Operation

```
XReg vaddr = X[rs1];
Bits<16> asid = X[rs2][ASID_WIDTH - 1:0];
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (misa.H == 1 && mode() == PrivilegeMode::VU) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
}
if (mstatus.TVM == 1 && mode() == PrivilegeMode::S) || (mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (misa.H == 1 && hstatus.VTVM == 1 && mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
if (!implemented?(ExtensionName::Sv32) && !implemented?(ExtensionName::Sv39) && !implemented?(ExtensionName::Sv48) &&
!implemented?(ExtensionName::Sv57)) {
  if (TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
VmaOrderType vma_type;
if (misa.H == 1 && mode() == PrivilegeMode::VS) {
  vma_type.vsmode = true;
  vma_type.single_vmid = true;
  vma_type.vmid = hgatp.VMID;
} else {
  vma_type.smode = true;
}
if ((rs1 == 0) && (rs2 == 0)) {
  vma_type.global = true;
  order_pgtbl_writes_before_vmafence(vma_type);
  invalidate_translations(vma_type);
  order_pgtbl_reads_after_vmafence(vma_type);
} else if ((rs1 == 0) && (rs2 != 0)) {
  vma_type.single_asid = true;
  vma_type.asid = asid;
  order_pgtbl_writes_before_vmafence(vma_type);
  invalidate_translations(vma_type);
  order_pgtbl_reads_after_vmafence(vma_type);
} else if ((rs1 != 0) && (rs2 == 0)) {
  if (canonical_vaddr?(vaddr)) {
    vma_type.single_vaddr = true;
    vma_type.vaddr = vaddr;
    order_pgtbl_writes_before_vmafence(vma_type);
    invalidate_translations(vma_type);
    order_pgtbl_reads_after_vmafence(vma_type);
  }
} else {
  if (canonical_vaddr?(vaddr)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    vma_type.single_vaddr = true;
    vma_type.vaddr = vaddr;
    order_pgtbl_writes_before_vmafence(vma_type);
    invalidate_translations(vma_type);
    order_pgtbl_reads_after_vmafence(vma_type);
  }
}
```

B.91.6. Sail Operation

```
{
  let addr : option(xlenbits) = if rs1 == 0b00000 then None() else Some(X(rs1));
  let asid : option(xlenbits) = if rs2 == 0b00000 then None() else Some(X(rs2));
  match cur_privilege {
    User       => { handle_illegal(); RETIRE_FAIL },
    Supervisor => match (architecture(get_mstatus_SXL(mstatus)), mstatus.TVM()) {
                    (Some(_), 0b1)  => { handle_illegal(); RETIRE_FAIL },
                    (Some(_), 0b0) => { flush_TLB(asid, addr); RETIRE_SUCCESS },
                    (_, _)           => internal_error(__FILE__, __LINE__, "unimplemented sfence architecture")
                  },
    Machine    => { flush_TLB(asid, addr); RETIRE_SUCCESS }
  }
}
```

B.91.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

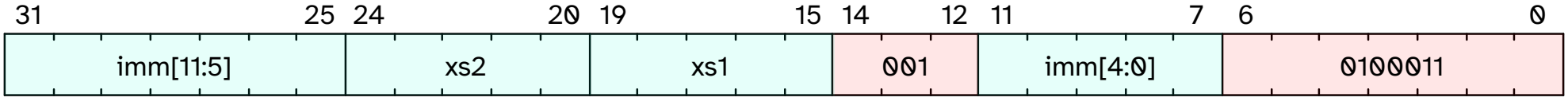
B.92. sh

Store halfword

This instruction is defined by:

- I, version >= I@2.1.0

B.92.1. Encoding



B.92.2. Description

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

B.92.3. Access

M	S	U
Always	Always	Always

B.92.4. Decode Variables

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

B.92.5. IDL Operation

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

B.92.6. 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.92.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

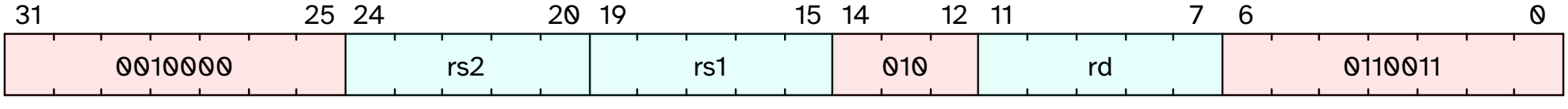
B.93. sh1add

Shift left by 1 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.93.1. Encoding



B.93.2. Description

Shifts `rs1` to the left by 1 bit and adds it to `rs2`.

B.93.3. Access

M	S	U
Always	Always	Always

B.93.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.93.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 1);
```

B.93.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_SH1ADD => 0b01,
    RISCV_SH2ADD => 0b10,
    RISCV_SH3ADD => 0b11
  };
  let result : xlenbits = (rs1_val << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.93.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

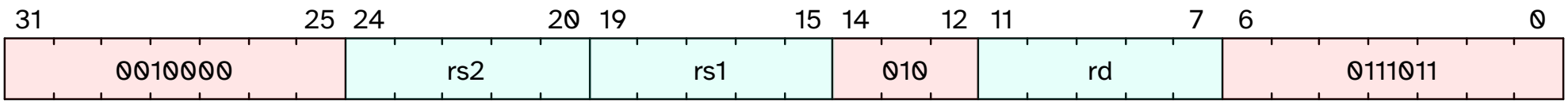
B.94. sh1add.uw

Shift unsigned word left by 1 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.94.1. Encoding



B.94.2. Description

Performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 1 place.

B.94.3. Access

M	S	U
Always	Always	Always

B.94.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.94.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 1);
```

B.94.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_ADDUW    => 0b00,
    RISCV_SH1ADDUW => 0b01,
    RISCV_SH2ADDUW => 0b10,
    RISCV_SH3ADDUW => 0b11
  };
  let result : xlenbits = (zero_extend(rs1_val[31..0]) << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.94.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

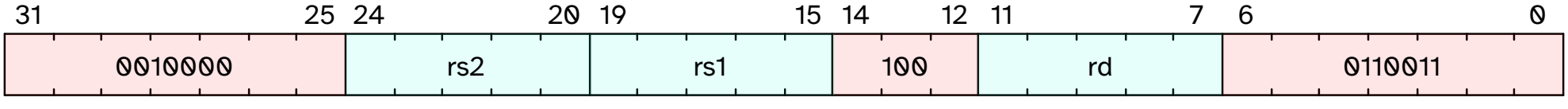
B.95. sh2add

Shift left by 2 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.95.1. Encoding



B.95.2. Description

Shifts `rs1` to the left by 2 places and adds it to `rs2`.

B.95.3. Access

M	S	U
Always	Always	Always

B.95.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.95.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 2);
```

B.95.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_SH1ADD => 0b01,
    RISCV_SH2ADD => 0b10,
    RISCV_SH3ADD => 0b11
  };
  let result : xlenbits = (rs1_val << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.95.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

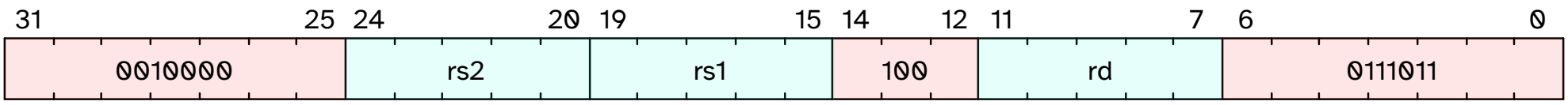
B.96. sh2add.uw

Shift unsigned word left by 2 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.96.1. Encoding



B.96.2. Description

Performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 2 places.

B.96.3. Access

M	S	U
Always	Always	Always

B.96.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.96.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 2);
```

B.96.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_ADDUW    => 0b00,
    RISCV_SH1ADDUW => 0b01,
    RISCV_SH2ADDUW => 0b10,
    RISCV_SH3ADDUW => 0b11
  };
  let result : xlenbits = (zero_extend(rs1_val[31..0]) << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.96.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

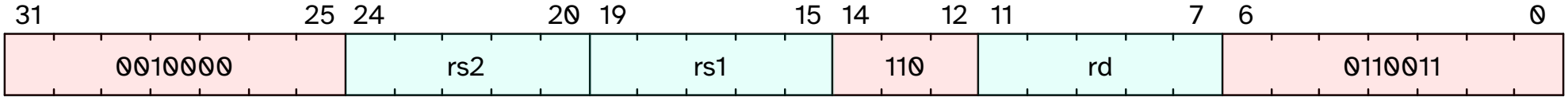
B.97. sh3add

Shift left by 3 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.97.1. Encoding



B.97.2. Description

Shifts `rs1` to the left by 3 places and adds it to `rs2`.

B.97.3. Access

M	S	U
Always	Always	Always

B.97.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.97.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1] << 3);
```

B.97.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_SH1ADD => 0b01,
    RISCV_SH2ADD => 0b10,
    RISCV_SH3ADD => 0b11
  };
  let result : xlenbits = (rs1_val << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.97.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

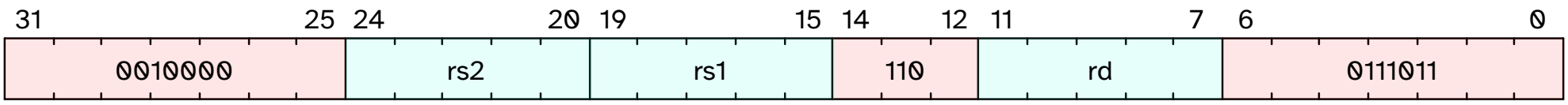
B.98. sh3add.uw

Shift unsigned word left by 3 and add

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.98.1. Encoding



B.98.2. Description

Performs an XLEN-wide addition of two addends. The first addend is rs2. The second addend is the unsigned value formed by extracting the least-significant word of rs1 and shifting it left by 3 places.

B.98.3. Access

M	S	U
Always	Always	Always

B.98.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.98.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs2] + (X[rs1][31:0] << 3);
```

B.98.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let shamt : bits(2) = match op {
    RISCV_ADDUW    => 0b00,
    RISCV_SH1ADDUW => 0b01,
    RISCV_SH2ADDUW => 0b10,
    RISCV_SH3ADDUW => 0b11
  };
  let result : xlenbits = (zero_extend(rs1_val[31..0]) << shamt) + rs2_val;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.98.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.99. sll

Shift left logical

This instruction is defined by:

- I, version >= I@2.1.0

B.99.1. Encoding

B.99.2. Description

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

B.99.3. Access

M	S	U
Always	Always	Always

B.99.4. Decode Variables

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

B.99.5. IDL Operation

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

B.99.6. 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.99.7. Exceptions

This instruction does not generate synchronous exceptions.

B.100. slli

Shift left logical immediate

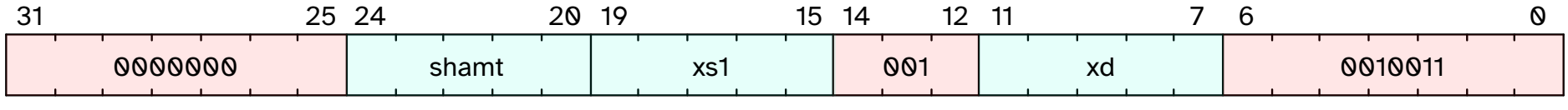
This instruction is defined by:

- I, version >= I@2.1.0

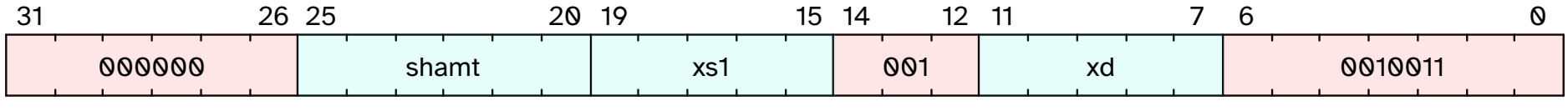
B.100.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.100.2. Description

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

B.100.3. Access

M	S	U
Always	Always	Always

B.100.4. 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.100.5. IDL Operation

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

B.100.6. 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.100.7. Exceptions

This instruction does not generate synchronous exceptions.

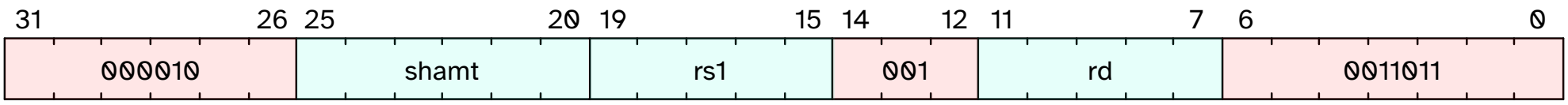
B.101. slli.uw

Shift left unsigned word (Immediate)

This instruction is defined by:

- Zba, version >= Zba@1.0.0

B.101.1. Encoding



B.101.2. Description

Takes the least-significant word of rs1, zero-extends it, and shifts it left by the immediate.

This instruction is the same as slli with `zext.w` performed on rs1 before shifting.

B.101.3. Access

M	S	U
Always	Always	Always

B.101.4. Decode Variables

```
Bits<6> shamt = $encoding[25:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.101.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = X[rs1][31:0] << shamt;
```

B.101.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let result : xlenbits = zero_extend(rs1_val[31..0]) << shamt;
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.101.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

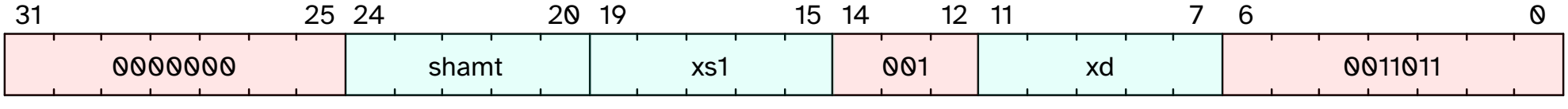
B.102. slliw

Shift left logical immediate word

This instruction is defined by:

- I, version >= I@2.1.0

B.102.1. Encoding



B.102.2. Description

Shift the 32-bit value in xs1 left by shamt, and store the sign-extended result in xd

B.102.3. Access

M	S	U
Always	Always	Always

B.102.4. Decode Variables

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

B.102.5. IDL Operation

```
X[xd] = sext(X[xs1] << shamt, 31);
```

B.102.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.102.7. Exceptions

This instruction does not generate synchronous exceptions.

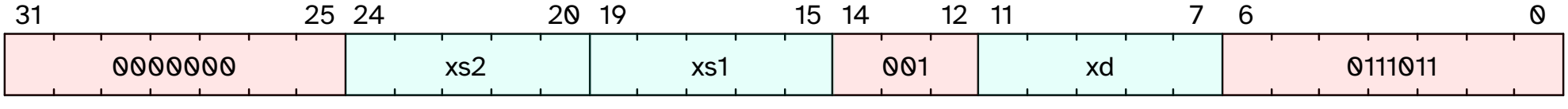
B.103. sllw

Shift left logical word

This instruction is defined by:

- I, version >= I@2.1.0

B.103.1. Encoding



B.103.2. Description

Shift the 32-bit value in `xs1` left by the value in the lower 5 bits of `xs2`, and store the sign-extended result in `xd`.

B.103.3. Access

M	S	U
Always	Always	Always

B.103.4. Decode Variables

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

B.103.5. IDL Operation

```
X[xd] = sext(X[xs1] << X[xs2][4:0], 31);
```

B.103.6. Sail Operation

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

B.103.7. Exceptions

This instruction does not generate synchronous exceptions.

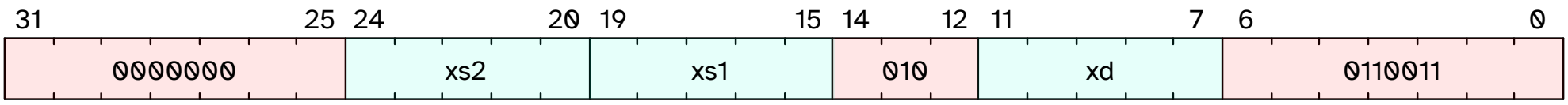
B.104. slt

Set on less than

This instruction is defined by:

- I, version >= I@2.1.0

B.104.1. Encoding



B.104.2. 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.104.3. Access

M	S	U
Always	Always	Always

B.104.4. Decode Variables

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

B.104.5. IDL Operation

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

B.104.6. 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.104.7. Exceptions

This instruction does not generate synchronous exceptions.

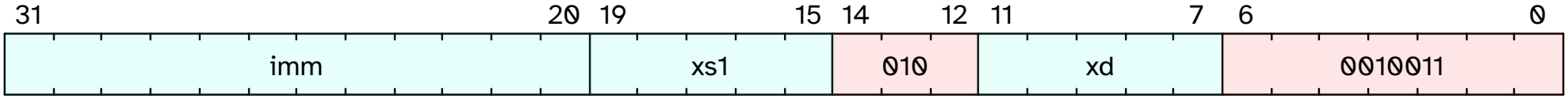
B.105. slti

Set on less than immediate

This instruction is defined by:

- I, version >= I@2.1.0

B.105.1. Encoding



B.105.2. 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.105.3. Access

M	S	U
Always	Always	Always

B.105.4. Decode Variables

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

B.105.5. IDL Operation

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

B.105.6. 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.105.7. Exceptions

This instruction does not generate synchronous exceptions.

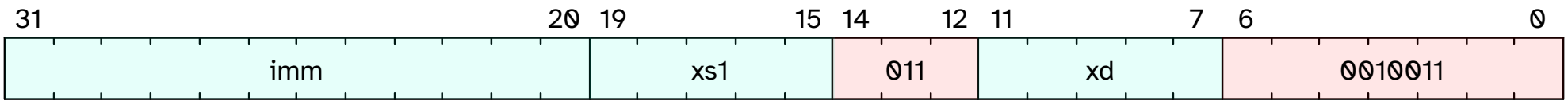
B.106. sltiu

Set on less than immediate unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.106.1. Encoding



B.106.2. 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.106.3. Access

M	S	U
Always	Always	Always

B.106.4. Decode Variables

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

B.106.5. IDL Operation

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

B.106.6. 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.106.7. Exceptions

This instruction does not generate synchronous exceptions.

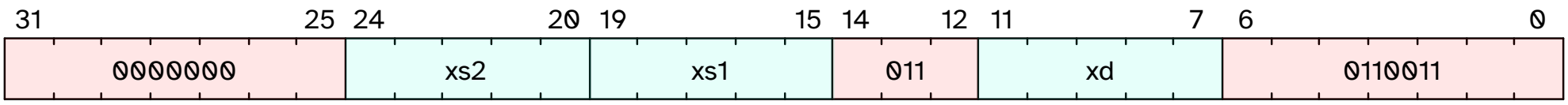
B.107. sltu

Set on less than unsigned

This instruction is defined by:

- I, version >= I@2.1.0

B.107.1. Encoding



B.107.2. 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.107.3. Access

M	S	U
Always	Always	Always

B.107.4. Decode Variables

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

B.107.5. IDL Operation

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

B.107.6. 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.107.7. Exceptions

This instruction does not generate synchronous exceptions.

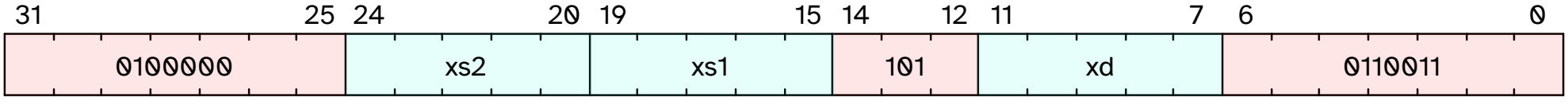
B.108. sra

Shift right arithmetic

This instruction is defined by:

- I, version >= I@2.1.0

B.108.1. Encoding



B.108.2. 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.108.3. Access

M	S	U
Always	Always	Always

B.108.4. Decode Variables

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

B.108.5. IDL Operation

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

B.108.6. 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.108.7. Exceptions

This instruction does not generate synchronous exceptions.

B.109. srai

Shift right arithmetic immediate

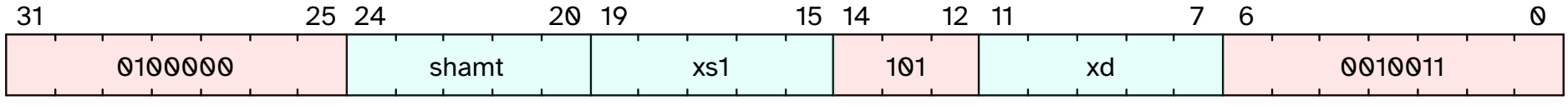
This instruction is defined by:

- I, version >= I@2.1.0

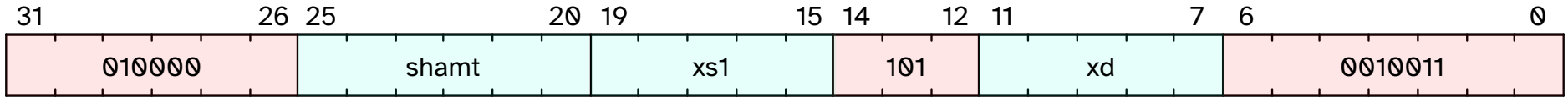
B.109.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.109.2. 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.109.3. Access

M	S	U
Always	Always	Always

B.109.4. 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.109.5. IDL Operation

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

B.109.6. 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.109.7. Exceptions

This instruction does not generate synchronous exceptions.

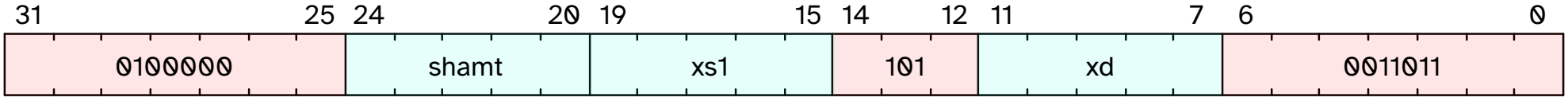
B.110. sraiw

Shift right arithmetic immediate word

This instruction is defined by:

- I, version >= I@2.1.0

B.110.1. Encoding



B.110.2. Description

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the 32-bit value in xs1 right by shamt, and store the sign-extended result in xd.

B.110.3. Access

M	S	U
Always	Always	Always

B.110.4. Decode Variables

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

B.110.5. IDL Operation

```
XReg operand = sext(X[xs1], 31);
X[xd] = sext(operand >>> shamt, 31);
```

B.110.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.110.7. Exceptions

This instruction does not generate synchronous exceptions.

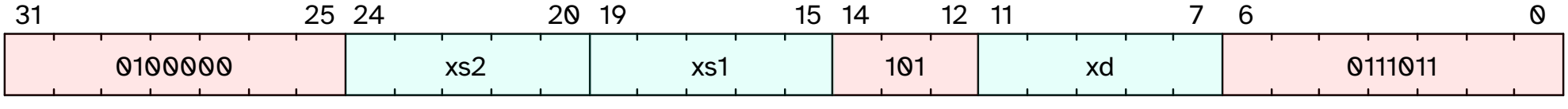
B.111. sraw

Shift right arithmetic word

This instruction is defined by:

- I, version >= I@2.1.0

B.111.1. Encoding



B.111.2. Description

Arithmetic shift the 32-bit value in `xs1` right by the value in the lower 5 bits of `xs2`, and store the sign-extended result in `xd`.

B.111.3. Access

M	S	U
Always	Always	Always

B.111.4. Decode Variables

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

B.111.5. IDL Operation

```
XReg operand1 = sext(X[xs1], 31);
X[xd] = sext(operand1 >>> X[xs2][4:0], 31);
```

B.111.6. Sail Operation

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

B.111.7. Exceptions

This instruction does not generate synchronous exceptions.

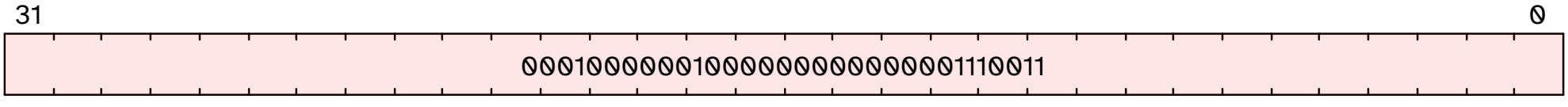
B.112. sret

Supervisor Exception Return

This instruction is defined by:

- S, version >= S@1.11.0

B.112.1. Encoding



B.112.2. Description

Returns from an exception.

When sret is allowed to execute, its behavior depends on whether or not the current privilege mode is virtualized.

When the current privilege mode is (H)S-mode or M-mode

sret sets hstatus = 0, mstatus.SPP = 0, mstatus.SIE = mstatus.SPIE, and mstatus.SPIE = 1, changes the privilege mode according to the table below, and then jumps to the address in sepc.

Table 9. Next privilege mode following an sret in (H)S-mode or M-mode

mstatus.SPP	hstatus.SPV	Mode after sret
0	0	U-mode
0	1	VU-mode
1	0	(H)S-mode
1	1	VS-mode

When the current privilege mode is VS-mode

sret sets vsstatus.SPP = 0, vsstatus.SIE = `vstatus.SPIE`, and vsstatus.SPIE = 1, changes the privilege mode according to the table below, and then jumps to the address in vsepc.

Table 10. Next privilege mode following an sret in (H)S-mode or M-mode

vsstatus.SPP	Mode after sret
0	VU-mode
1	VS-mode

B.112.3. Access

M	S	U
Always	Sometimes	Never

Access is determined as follows:

mstatus.TSR	hstatus.VTSR	Behavior when executed from:				
		M-mode	U-mode	(H)S-mode	VU-mode	VS-mode
0	0	executes	`Illegal Instruction`	executes	`Virtual Instruction`	executes
0	1	executes	`Illegal Instruction`	executes	`Virtual Instruction`	`Virtual Instruction`
1	0	executes	`Illegal Instruction`	`Illegal Instruction`	`Virtual Instruction`	executes
1	1	executes	`Illegal Instruction`	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`

B.112.4. Decode Variables

```
```

B.112.5. IDL Operation

```
if (implemented?(ExtensionName::H)) {
  if (mstatus.TSR == 1'b0 && hstatus.VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
    } else if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (mstatus.TSR == 1'b0 && hstatus.VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode::VS) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (mstatus.TSR == 1'b1 && hstatus.VTSR == 1'b0) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  } else if (mstatus.TSR == 1'b1 && hstatus.VTSR == 1'b1) {
    if (mode() == PrivilegeMode::U || mode() == PrivilegeMode::S) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    } else if (mode() == PrivilegeMode::VU || mode() == PrivilegeMode::VS) {
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  }
} else {
  if (mode() != PrivilegeMode::U) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
if (!virtual_mode?()) {
  if (implemented?(ExtensionName::H)) {
    if (hstatus.SPV == 1'b1) {
      if (mstatus.SPP == 2'b01) {
        set_mode(PrivilegeMode::VS);
      } else if (mstatus.SPP == 2'b00) {
        set_mode(PrivilegeMode::VU);
      }
    }
    hstatus.SPV = 0;
  }
  mstatus.SIE = mstatus.SPIE;
  mstatus.SPIE = 1;
  mstatus.SPP = 2'b00;
  $pc = $bits(sepc);
} else {
  if (mstatus.TSR == 1'b1) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
  vsstatus.SPP = 0;
  vsstatus.SIE = vsstatus.SPIE;
  vsstatus.SPIE = 1;
  $pc = $bits(vsepc);
}
```

B.112.6. Sail Operation

```
{
  let sret_illegal : bool = match cur_privilege {
    User       => true,
    Supervisor => not(haveSupMode ()) | mstatus.TSR() == 0b1,
    Machine    => not(haveSupMode ())
  };
  if   sret_illegal
  then { handle_illegal(); RETIRE_FAIL }
  else if not(ext_check_xret_priv (Supervisor))
  then { ext_fail_xret_priv(); RETIRE_FAIL }
  else {
    set_next_pc(exception_handler(cur_privilege, CTL_SRET(), PC));
    RETIRE_SUCCESS
  }
}
```

B.112.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

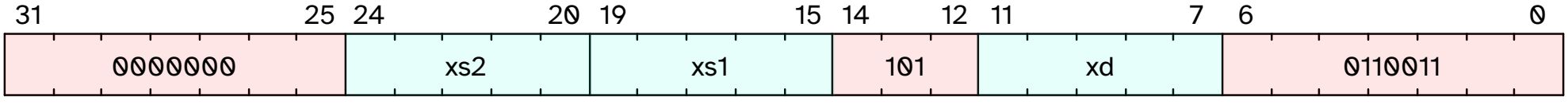
B.113. srl

Shift right logical

This instruction is defined by:

- I, version >= I@2.1.0

B.113.1. Encoding



B.113.2. Description

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

B.113.3. Access

M	S	U
Always	Always	Always

B.113.4. Decode Variables

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

B.113.5. IDL Operation

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

B.113.6. 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.113.7. Exceptions

This instruction does not generate synchronous exceptions.

B.114. srli

Shift right logical immediate

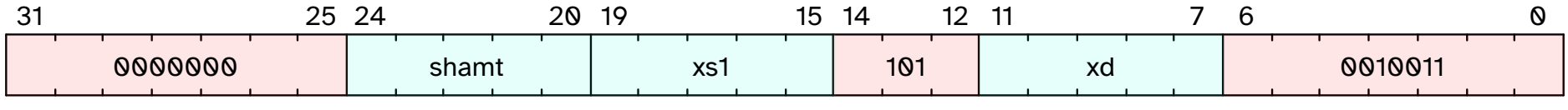
This instruction is defined by:

- I, version >= I@2.1.0

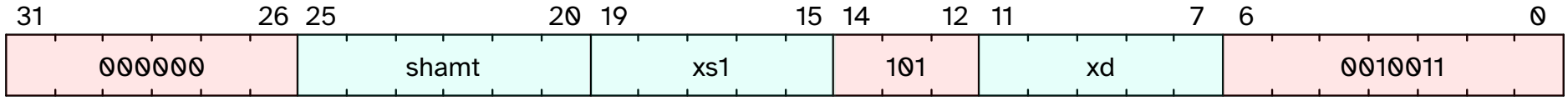
B.114.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.114.2. Description

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

B.114.3. Access

M	S	U
Always	Always	Always

B.114.4. 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.114.5. IDL Operation

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

B.114.6. 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.114.7. Exceptions

This instruction does not generate synchronous exceptions.

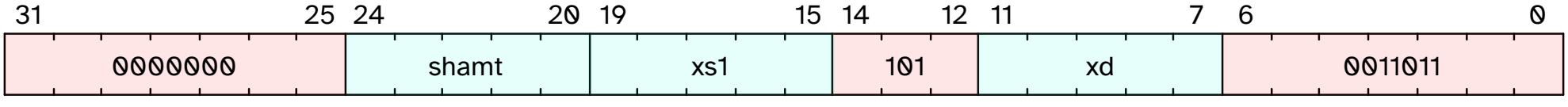
B.115. srliw

Shift right logical immediate word

This instruction is defined by:

- I, version >= I@2.1.0

B.115.1. Encoding



B.115.2. Description

Shift the 32-bit value in xs1 right by shamt, and store the sign-extended result in xd

B.115.3. Access

M	S	U
Always	Always	Always

B.115.4. Decode Variables

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

B.115.5. IDL Operation

```
XReg operand = X[xs1][31:0];
X[xd] = sext(operand >> shamt, 31);
```

B.115.6. Sail Operation

```
{
  let xs1_val = (X(xs1))[31..0];
  let result : bits(32) = match op {
    RISCV_SLLIW => xs1_val << shamt,
    RISCV_SRLIW => xs1_val >> shamt,
    RISCV_SRAIW => shift_right_arith32(xs1_val, shamt)
  };
  X(xd) = sign_extend(result);
  RETIRE_SUCCESS
}
```

B.115.7. Exceptions

This instruction does not generate synchronous exceptions.

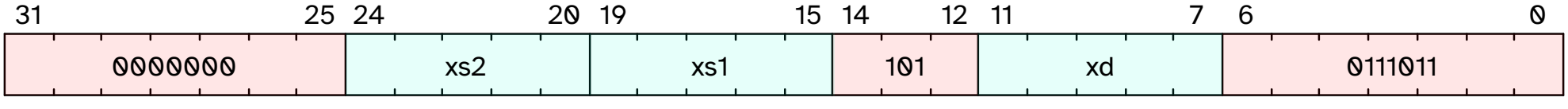
B.116. srlw

Shift right logical word

This instruction is defined by:

- I, version >= I@2.1.0

B.116.1. Encoding



B.116.2. Description

Logical shift the 32-bit value in `xs1` right by the value in the lower 5 bits of `xs2`, and store the sign-extended result in `xd`.

B.116.3. Access

M	S	U
Always	Always	Always

B.116.4. Decode Variables

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

B.116.5. IDL Operation

```
X[xd] = sext(X[xs1][31:0] >> X[xs2][4:0], 31);
```

B.116.6. Sail Operation

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

B.116.7. Exceptions

This instruction does not generate synchronous exceptions.

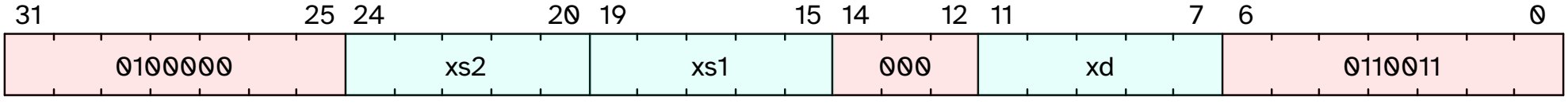
B.117. sub

Subtract

This instruction is defined by:

- I, version >= I@2.1.0

B.117.1. Encoding



B.117.2. Description

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

B.117.3. Access

M	S	U
Always	Always	Always

B.117.4. Decode Variables

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

B.117.5. IDL Operation

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

B.117.6. 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.117.7. Exceptions

This instruction does not generate synchronous exceptions.

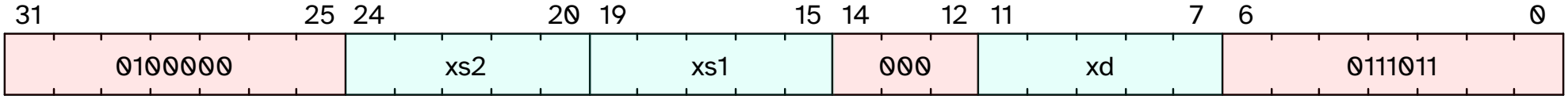
B.118. subw

Subtract word

This instruction is defined by:

- I, version >= I@2.1.0

B.118.1. Encoding



B.118.2. Description

Subtract the 32-bit values in xs2 from xs1, and store the sign-extended result in xd

B.118.3. Access

M	S	U
Always	Always	Always

B.118.4. Decode Variables

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

B.118.5. IDL Operation

```
Bits<32> t0 = X[xs1][31:0];
Bits<32> t1 = X[xs2][31:0];
X[xd] = sext(t0 - t1, 31);
```

B.118.6. Sail Operation

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

B.118.7. Exceptions

This instruction does not generate synchronous exceptions.

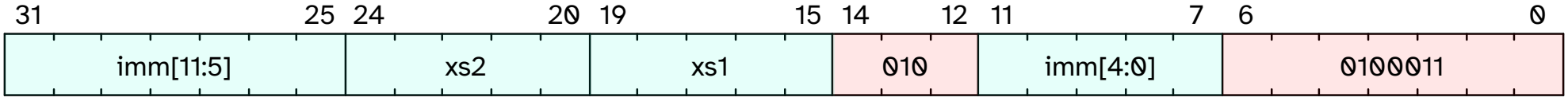
B.119. sw

Store word

This instruction is defined by:

- I, version >= I@2.1.0

B.119.1. Encoding



B.119.2. Description

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

B.119.3. Access

M	S	U
Always	Always	Always

B.119.4. Decode Variables

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

B.119.5. IDL Operation

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

B.119.6. 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.119.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

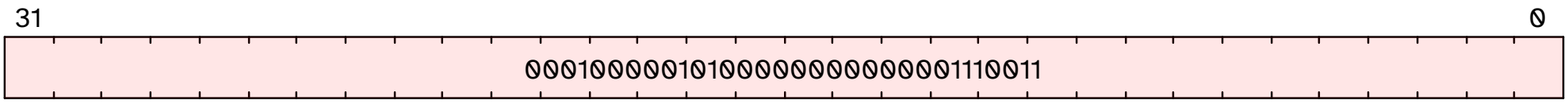
B.120. wfi

Wait for interrupt

This instruction is defined by:

- Sm, version >= Sm@1.11.0

B.120.1. Encoding



B.120.2. 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 -%>

B.120.3. Access

M	S	U
Always	Sometimes	Sometimes

<%- 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.120.4. Decode Variables

B.120.5. IDL Operation

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

B.120.6. 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.120.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

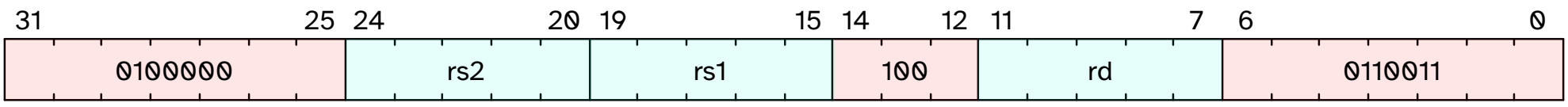
B.121. xnor

Exclusive NOR

This instruction is defined by:

- anyOf:
  - Zbb, version >= Zbb@1.0.0
  - Zbkb, version >= Zbkb@1.0.0

B.121.1. Encoding



B.121.2. Description

Performs the bit-wise exclusive-NOR operation on rs1 and rs2.

B.121.3. Access

M	S	U
Always	Always	Always

B.121.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.121.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (misa.B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = ~(X[rs1] ^ X[rs2]);
```

B.121.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(rs2);
  let result : xlenbits = match op {
    RISCV_ANDN => rs1_val & ~(rs2_val),
    RISCV_ORN  => rs1_val | ~(rs2_val),
    RISCV_XNOR => ~(rs1_val ^ rs2_val),
    RISCV_MAX  => to_bits(sizeof(xlen), max(signed(rs1_val),   signed(rs2_val))),
    RISCV_MAXU => to_bits(sizeof(xlen), max(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_MIN  => to_bits(sizeof(xlen), min(signed(rs1_val),   signed(rs2_val))),
    RISCV_MINU => to_bits(sizeof(xlen), min(unsigned(rs1_val), unsigned(rs2_val))),
    RISCV_ROL  => if sizeof(xlen) == 32
                  then rs1_val <<< rs2_val[4..0]
                  else rs1_val <<< rs2_val[5..0],
    RISCV_ROR  => if sizeof(xlen) == 32
                  then rs1_val >>> rs2_val[4..0]
                  else rs1_val >>> rs2_val[5..0]
  };
  X(rd) = result;
  RETIRE_SUCCESS
}
```

B.121.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

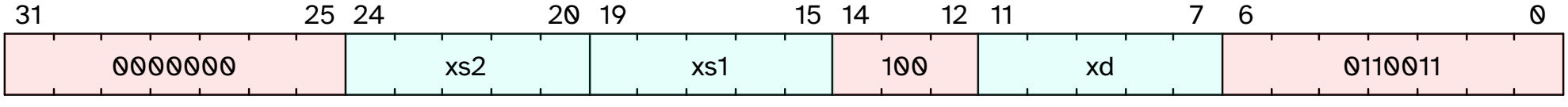
B.122. xor

Exclusive Or

This instruction is defined by:

- I, version >= I@2.1.0

B.122.1. Encoding



B.122.2. Description

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

B.122.3. Access

M	S	U
Always	Always	Always

B.122.4. Decode Variables

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

B.122.5. IDL Operation

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

B.122.6. 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.122.7. Exceptions

This instruction does not generate synchronous exceptions.

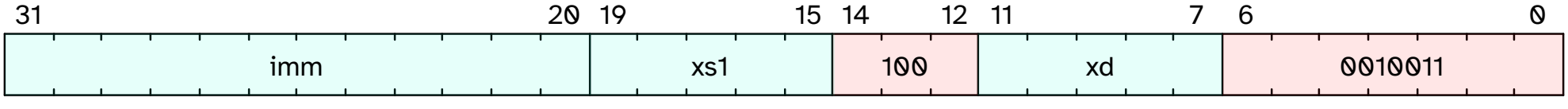
B.123. xori

Exclusive Or immediate

This instruction is defined by:

- I, version >= I@2.1.0

B.123.1. Encoding



B.123.2. Description

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

B.123.3. Access

M	S	U
Always	Always	Always

B.123.4. Decode Variables

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

B.123.5. IDL Operation

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

B.123.6. 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.123.7. Exceptions

This instruction does not generate synchronous exceptions.

Appendix C: CSR Details

C.1. cycle

Cycle counter for RDCYCLE Instruction

Alias for M-mode CSR mcycle.

Privilege mode access is controlled with mcounteren.CY, scounteren.CY, and hcounteren.CY as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.1.1. Attributes

CSR Address	0xc00
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

C.1.2. Format

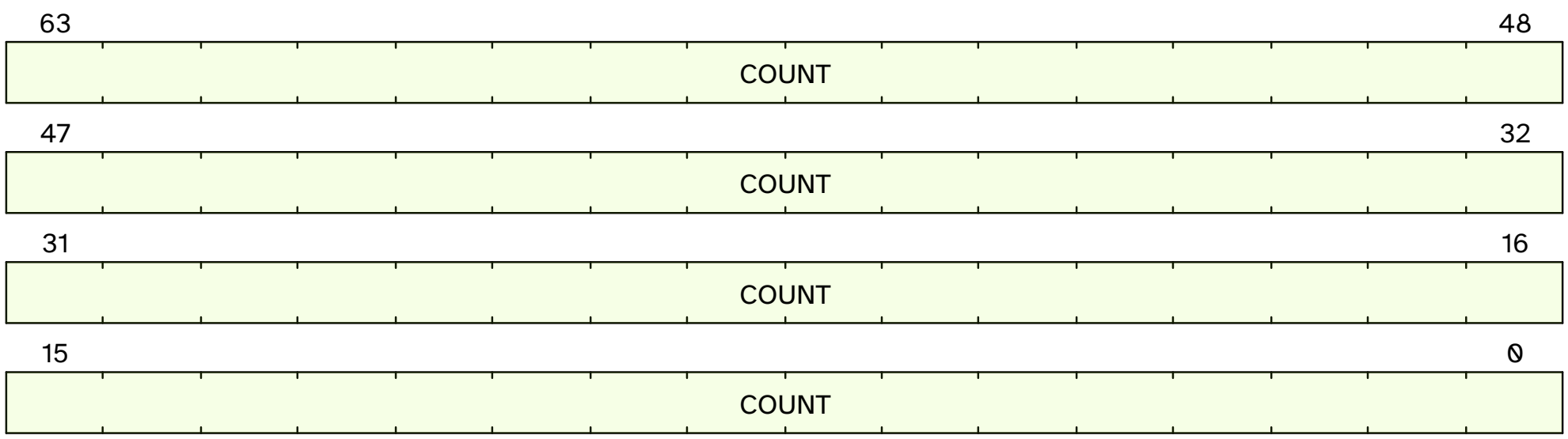


Figure 1. cycle format

C.1.3. Field Summary

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

C.1.4. Fields

cycle.COUNT Field

Location:

63:0

Description:

Alias of mcycle.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.1.5. Software read

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

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

C.2. cycleh

High-half cycle counter for RDCYCLE Instruction

cycleh is only defined in RV32.

Alias for M-mode CSR mcycleh.

Privilege mode access is controlled with mcounteren.CY, scounteren.CY, and hcounteren.CY as follows:

mcounteren.CY	scounteren.CY	hcounteren.CY	cycle behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.2.1. Attributes

CSR Address	0xc80
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

C.2.2. Format

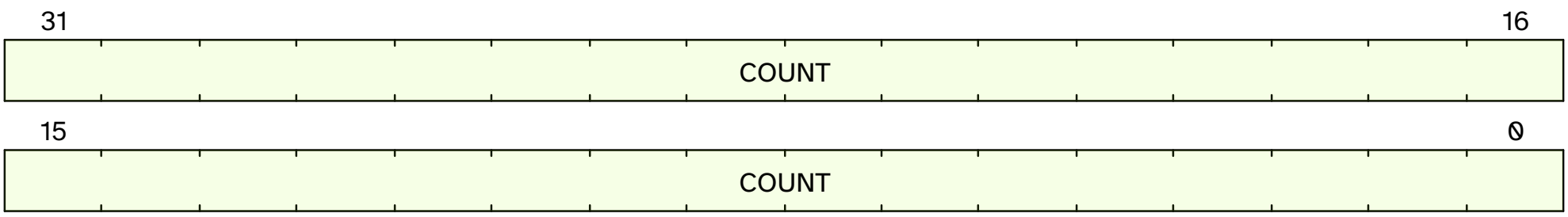


Figure 2. cycleh format

C.2.3. Field Summary

Name	Location	Type	Reset Value
cycleh.COUNT	31:0	RO-H	UNDEFINED_LEGAL

C.2.4. Fields

cycleh.COUNT Field

Location:

31:0

Description:

Alias of mcycleh.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.2.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.CY == 1'b0) {
```

```
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.CY & scounteren.CY) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.CY == 1'b0 && mcounteren.CY == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.CY & scounteren.CY) == 1'b0) && (mcounteren.CY == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle()[63:32];
```

C.3. 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.3.1. Attributes

CSR Address	0xc02
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

C.3.2. Format

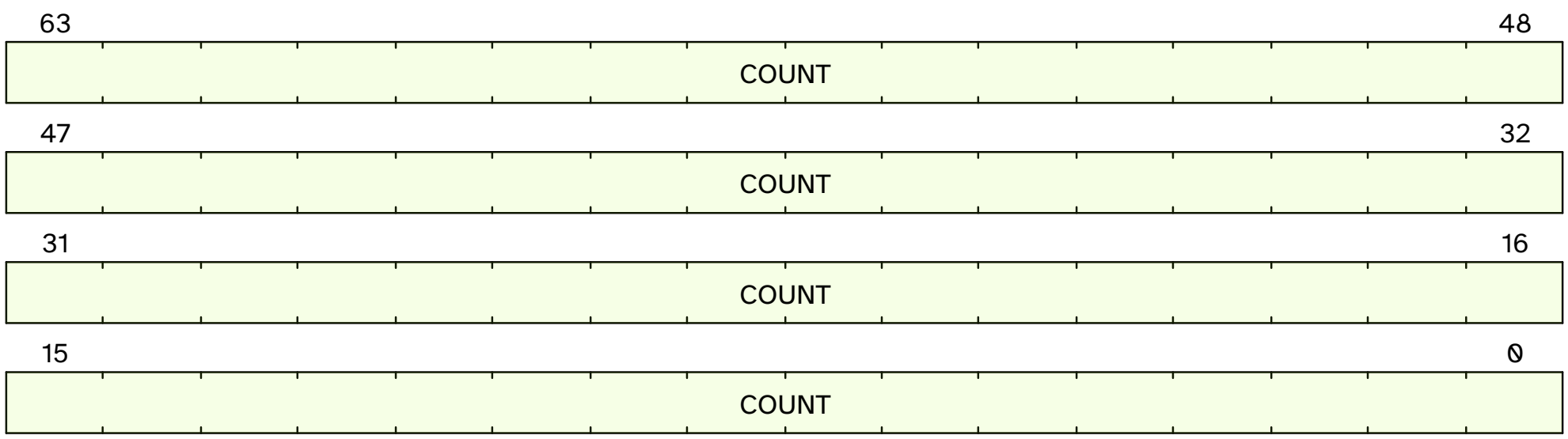


Figure 3. instret format

C.3.3. Field Summary

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

C.3.4. Fields

instret.COUNT Field

Location:

63:0

Description:

Alias of minstret.COUNT.

Type:

RO-H

Reset value:

0

C.3.5. Software read

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

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

C.4. instreth

Instructions retired counter, high bits

instreth is only defined in RV32.

Alias for high bits of M-mode CSR minstret[63:32].

Privilege mode access is controlled with mcounteren.IR, scounteren.IR, and hcounteren.IR as follows:

mcounteren.IR	scounteren.IR	hcounteren.IR	instret behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	IllegalInstruction	IllegalInstruction	IllegalInstruction	IllegalInstruction
1	0	0	read-only	IllegalInstruction	VirtualInstruction	VirtualInstruction
1	1	0	read-only	read-only	VirtualInstruction	VirtualInstruction
1	0	1	read-only	IllegalInstruction	read-only	VirtualInstruction
1	1	1	read-only	read-only	read-only	read-only

C.4.1. Attributes

CSR Address	0xc82
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

C.4.2. Format

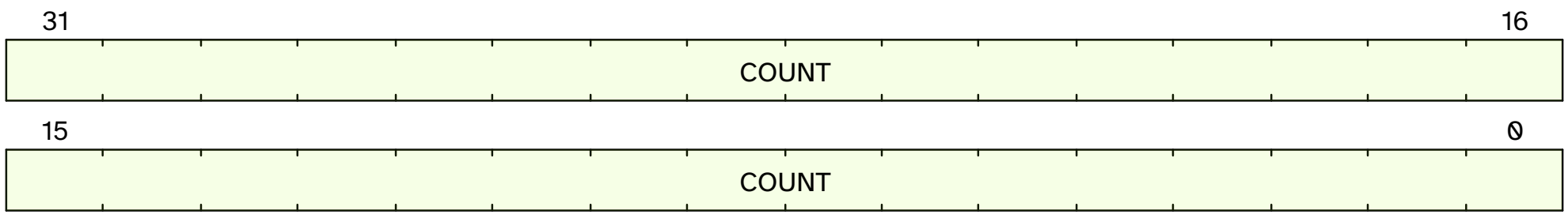


Figure 4. instreth format

C.4.3. Field Summary

Name	Location	Type	Reset Value
instreth.COUNT	31:0	RO-H	UNDEFINED_LEGAL

C.4.4. Fields

instreth.COUNT Field

Location:

31:0

Description:

Alias of minstret.COUNT[63:32].

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.4.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.IR == 1'b0) {
```

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

C.5. 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.5.1. Attributes

CSR Address	0xf12
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.5.2. Format

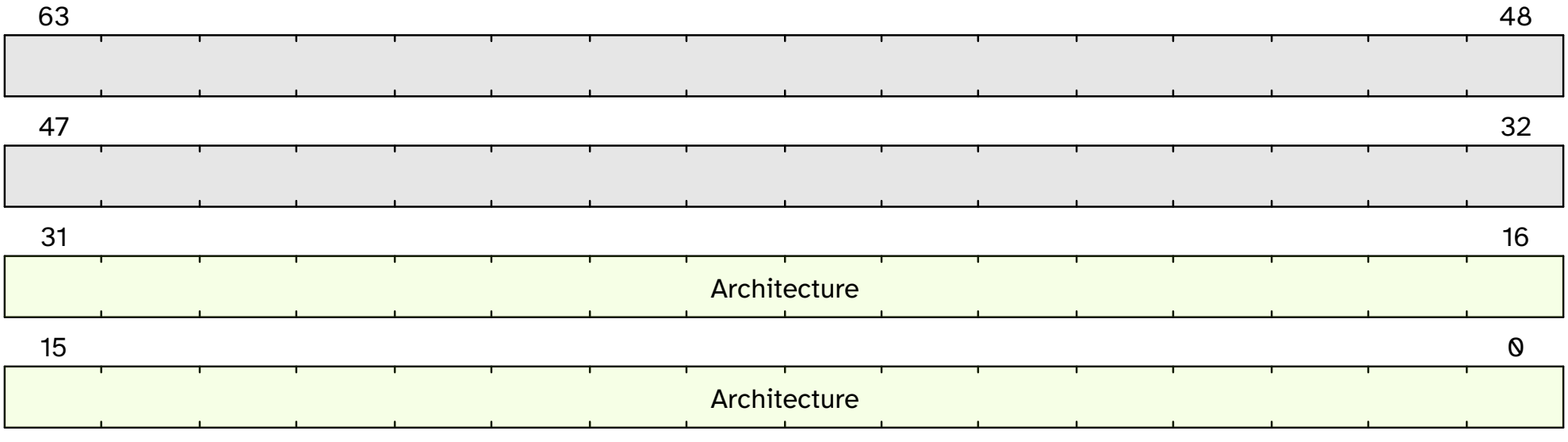


Figure 5. marchid format

C.5.3. Field Summary

Name	Location	Type	Reset Value
marchid.Architecture	63:0	RO	UNDEFINED_LEGAL

C.5.4. Fields

marchid.Architecture Field

Location:

63:0

Description:

Vendor-specific microarchitecture ID.

Type:

RO

Reset value:

UNDEFINED_LEGAL

C.6. mcause

Machine Cause

Reports the cause of the latest exception.

C.6.1. Attributes

CSR Address	0x342
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.6.2. Format

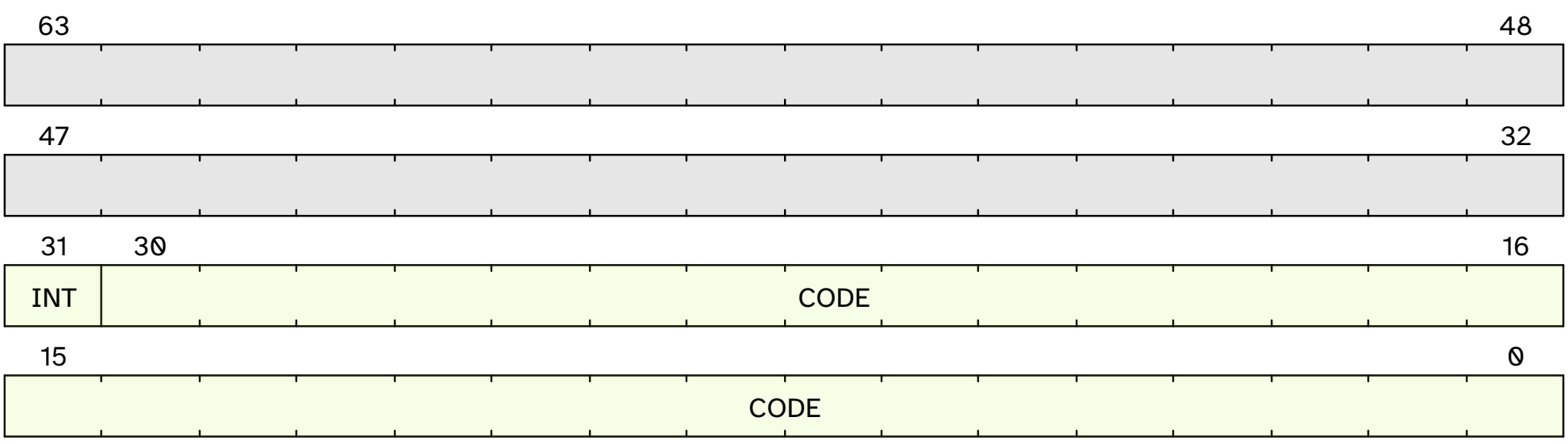


Figure 6. mcause format

C.6.3. Field Summary

Name	Location	Type	Reset Value
mcause.INT	63	RW-RH	0
mcause.CODE	62:0	RW-RH	0

C.6.4. Fields

mcause.INT Field

Location:

63

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:

62:0

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="!"]
!===
<%- interrupt_codes.sort_by{ |code| code.num }.each do |code| -%>
! <%= code.num %> ! <%= code.name %>
<%- end -%>
!===

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

Type:

RW-RH

Reset value:

`0`

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:

```
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.7. mcounteren

Machine Counter Enable

The counter-enable mcounteren register is a 32-bit register that controls the availability of the hardware performance-monitoring counters to <%- if ext?(:S) -%> S-mode <%- elsif ext?(:U) -%> U-mode <%- else -%> the next-lower privileged mode <%- end -%> .

The settings in this register only control accessibility. The act of reading or writing this register does not affect the underlying counters, which continue to increment even when not accessible.

When the CY, TM, IR, or HPMn bit in the mcounteren register is clear, attempts to read the cycle, time, instret, or `hpmcountern` register while executing in <%- if ext?(:S) -%> S-mode <%- elsif ext?(:U) -%> U-mode <%- else -%> S-mode or U-mode <%- end -%> will cause an `IllegalInstruction` exception. When one of these bits is set, access to the corresponding register is permitted in <%- if ext?(:S) -%> S-mode <%- elsif ext?(:U) -%> U-mode <%- else -%> the next implemented privilege mode (S-mode if implemented, otherwise U-mode). <%- end -%>

The counter-enable bits support two common use cases with minimal hardware. For harts that do not need high-performance timers and counters, machine-mode software can trap accesses and implement all features in software. For harts that need high-performance timers and counters but are not concerned with obfuscating the underlying hardware counters, the counters can be directly exposed to lower privilege modes.

The cycle, instret, and `hpmcountern` CSRs are read-only shadows of mcycle, minstret, and `mhpmcounter n`, respectively. The time CSR is a read-only shadow of the memory-mapped `mtime` register. <%- if possible_xlens.include?(32) -%> Analogously, on RV32I the cycleh, instreth and `hpmcounternh` CSRs are read-only shadows of mcycleh, minstreth and `mhpmcounternh`, respectively. On RV32I the timeh CSR is a read-only shadow of the upper 32 bits of the memory-mapped `mtime` register, while time shadows only the lower 32 bits of `mtime`. <%- end -%>

Implementations can convert reads of the time and timeh CSRs into loads to the memory-mapped `mtime` register, or emulate this functionality on behalf of less-privileged modes in M-mode software.

<%- if !ext?(:U) -%> In harts with U-mode, the mcounteren CSR must be implemented, but all fields are WARL and may be read-only zero, indicating reads to the corresponding counter will cause an `IllegalInstruction` exception when executing in a less-privileged mode. In harts without U-mode, the mcounteren register should not exist. <%- end -%>

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

The cycle, instret, and `hpmcountern` CSRs can also be made available to U-mode through the scounteren CSR <%- if ext?(:H) -%> and to VS-mode and/or VU-mode through hcounteren <%- end -%> . <%- end -%>

C.7.1. Attributes

CSR Address	0x306
Defining extension	• U, version >= U@1.0.0
Length	32-bit
Privilege Mode	M

C.7.2. Format

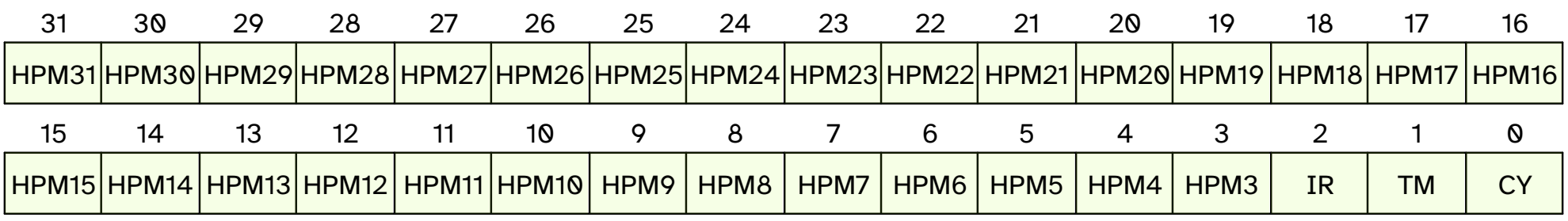


Figure 7. mcounteren format

C.7.3. Field Summary

Name	Location	Type	Reset Value
mcounteren.CY	0	RW RO	UNDEFINED_LEGAL
mcounteren.TM	1	RO	0
mcounteren.IR	2	RW RO	UNDEFINED_LEGAL
mcounteren.HPM3	3	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
mcounteren.HPM4	4	RW RO	UNDEFINED_LEGAL
mcounteren.HPM5	5	RW RO	UNDEFINED_LEGAL
mcounteren.HPM6	6	RW RO	UNDEFINED_LEGAL
mcounteren.HPM7	7	RW RO	UNDEFINED_LEGAL
mcounteren.HPM8	8	RW RO	UNDEFINED_LEGAL
mcounteren.HPM9	9	RW RO	UNDEFINED_LEGAL
mcounteren.HPM10	10	RW RO	UNDEFINED_LEGAL
mcounteren.HPM11	11	RW RO	UNDEFINED_LEGAL
mcounteren.HPM12	12	RW RO	UNDEFINED_LEGAL
mcounteren.HPM13	13	RW RO	UNDEFINED_LEGAL
mcounteren.HPM14	14	RW RO	UNDEFINED_LEGAL
mcounteren.HPM15	15	RW RO	UNDEFINED_LEGAL
mcounteren.HPM16	16	RW RO	UNDEFINED_LEGAL
mcounteren.HPM17	17	RW RO	UNDEFINED_LEGAL
mcounteren.HPM18	18	RW RO	UNDEFINED_LEGAL
mcounteren.HPM19	19	RW RO	UNDEFINED_LEGAL
mcounteren.HPM20	20	RW RO	UNDEFINED_LEGAL
mcounteren.HPM21	21	RW RO	UNDEFINED_LEGAL
mcounteren.HPM22	22	RW RO	UNDEFINED_LEGAL
mcounteren.HPM23	23	RW RO	UNDEFINED_LEGAL
mcounteren.HPM24	24	RW RO	UNDEFINED_LEGAL
mcounteren.HPM25	25	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
mcounteren.HPM26	26	RW RO	UNDEFINED_LEGAL
mcounteren.HPM27	27	RW RO	UNDEFINED_LEGAL
mcounteren.HPM28	28	RW RO	UNDEFINED_LEGAL
mcounteren.HPM29	29	RW RO	UNDEFINED_LEGAL
mcounteren.HPM30	30	RW RO	UNDEFINED_LEGAL
mcounteren.HPM31	31	RW RO	UNDEFINED_LEGAL

C.7.4. Fields

mcounteren.CY Field

Location:

0

Description:

When set, the cycle CSR (an alias of mcycle) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.CY is also set, cycle is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.CY is also set, cycle is further accessible to VS-mode.

When hcounteren.CY && scounteren.CY are both set, cycle is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.TM Field

Location:

1

Description:

Placeholder for delegating time to less-privileged modes; however, since time is memory-mapped rather than a CSR, this field is always read-only zero.

Type:

RO

Reset value:

0

mcounteren.IR Field

Location:

2

Description:

When set, the instret CSR (an alias of minstret) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.IR is also set, instret is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.IR is also set, instret is further accessible to VS-mode.

When hcounteren.IR && scounteren.IR are both set, instret is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM3 Field

Location:

3

Description:

When set, the hpmcounter3 CSR (an alias of mhpmcounter3) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM3 is also set, hpmcounter3 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM3 is also set, hpmcounter3 is further accessible to VS-mode.

When hcounteren.HPM3 && scounteren.HPM3 are both set, hpmcounter3 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM4 Field

Location:

4

Description:

When set, the hpmcounter4 CSR (an alias of mhpmcounter4) is accessible to

<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM4 is also set, hpmcounter4 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM4 is also set, hpmcounter4 is further accessible to VS-mode.

When hcounteren.HPM4 && scounteren.HPM4 are both set, hpmcounter4 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM5 Field

Location:

5

Description:

When set, the hpmcounter5 CSR (an alias of mhpmcounter5) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM5 is also set, hpmcounter5 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM5 is also set, hpmcounter5 is further accessible to VS-mode.

When hcounteren.HPM5 && scounteren.HPM5 are both set, hpmcounter5 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM6 Field

Location:

6

Description:

When set, the hpmcounter6 CSR (an alias of mhpmcounter6) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM6 is also set, hpmcounter6 is further accessible to U-mode.

<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM6 is also set, hpmcounter6 is further accessible to VS-mode.

When hcounteren.HPM6 && scounteren.HPM6 are both set, hpmcounter6 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM7 Field

Location:

7

Description:

When set, the hpmcounter7 CSR (an alias of mhpmcounter7) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM7 is also set, hpmcounter7 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM7 is also set, hpmcounter7 is further accessible to VS-mode.

When hcounteren.HPM7 && scounteren.HPM7 are both set, hpmcounter7 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM8 Field

Location:

8

Description:

When set, the hpmcounter8 CSR (an alias of mhpmcounter8) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM8 is also set, hpmcounter8 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM8 is also set, hpmcounter8 is further accessible to VS-mode.

When hcounteren.HPM8 && scounteren.HPM8 are both set, hpmcounter8 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM9 Field

Location:

9

Description:

When set, the hpmcounter9 CSR (an alias of mhpmcounter9) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM9 is also set, hpmcounter9 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM9 is also set, hpmcounter9 is further accessible to VS-mode.

When hcounteren.HPM9 && scounteren.HPM9 are both set, hpmcounter9 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM10 Field

Location:

10

Description:

When set, the hpmcounter10 CSR (an alias of mhpmcounter10) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM10 is also set, hpmcounter10 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM10 is also set, hpmcounter10 is further accessible to VS-mode.

When hcounteren.HPM10 && scounteren.HPM10 are both set, hpmcounter10 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM11 Field

Location:

11

Description:

When set, the hpmcounter11 CSR (an alias of mhpmcounter11) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM11 is also set, hpmcounter11 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM11 is also set, hpmcounter11 is further accessible to VS-mode.

When hcounteren.HPM11 && scounteren.HPM11 are both set, hpmcounter11 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM12 Field

Location:

12

Description:

When set, the hpmcounter12 CSR (an alias of mhpmcounter12) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM12 is also set, hpmcounter12 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM12 is also set, hpmcounter12 is further accessible to VS-mode.

When hcounteren.HPM12 && scounteren.HPM12 are both set, hpmcounter12 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM13 Field

Location:

13

Description:

When set, the hpmcounter13 CSR (an alias of mhpmcounter13) is accessible to

<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM13 is also set, hpmcounter13 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM13 is also set, hpmcounter13 is further accessible to VS-mode.

When hcounteren.HPM13 && scounteren.HPM13 are both set, hpmcounter13 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM14 Field

Location:

14

Description:

When set, the hpmcounter14 CSR (an alias of mhpmcounter14) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM14 is also set, hpmcounter14 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM14 is also set, hpmcounter14 is further accessible to VS-mode.

When hcounteren.HPM14 && scounteren.HPM14 are both set, hpmcounter14 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM15 Field

Location:

15

Description:

When set, the hpmcounter15 CSR (an alias of mhpmcounter15) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM15 is also set, hpmcounter15 is further accessible to U-mode.

<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM15 is also set, hpmcounter15 is further accessible to VS-mode.

When hcounteren.HPM15 && scounteren.HPM15 are both set, hpmcounter15 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM16 Field

Location:

16

Description:

When set, the hpmcounter16 CSR (an alias of mhpmcounter16) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM16 is also set, hpmcounter16 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM16 is also set, hpmcounter16 is further accessible to VS-mode.

When hcounteren.HPM16 && scounteren.HPM16 are both set, hpmcounter16 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM17 Field

Location:

17

Description:

When set, the hpmcounter17 CSR (an alias of mhpmcounter17) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM17 is also set, hpmcounter17 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM17 is also set, hpmcounter17 is further accessible to VS-mode.

When hcounteren.HPM17 && scounteren.HPM17 are both set, hpmcounter17 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM18 Field

Location:

18

Description:

When set, the hpmcounter18 CSR (an alias of mhpmcounter18) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM18 is also set, hpmcounter18 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM18 is also set, hpmcounter18 is further accessible to VS-mode.

When hcounteren.HPM18 && scounteren.HPM18 are both set, hpmcounter18 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM19 Field

Location:

19

Description:

When set, the hpmcounter19 CSR (an alias of mhpmcounter19) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM19 is also set, hpmcounter19 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM19 is also set, hpmcounter19 is further accessible to VS-mode.

When hcounteren.HPM19 && scounteren.HPM19 are both set, hpmcounter19 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM20 Field

Location:

20

Description:

When set, the hpmcounter20 CSR (an alias of mhpmcounter20) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM20 is also set, hpmcounter20 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM20 is also set, hpmcounter20 is further accessible to VS-mode.

When hcounteren.HPM20 && scounteren.HPM20 are both set, hpmcounter20 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM21 Field

Location:

21

Description:

When set, the hpmcounter21 CSR (an alias of mhpmcounter21) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM21 is also set, hpmcounter21 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM21 is also set, hpmcounter21 is further accessible to VS-mode.

When hcounteren.HPM21 && scounteren.HPM21 are both set, hpmcounter21 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM22 Field

Location:

22

Description:

When set, the hpmcounter22 CSR (an alias of mhpmcounter22) is accessible to

<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM22 is also set, hpmcounter22 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM22 is also set, hpmcounter22 is further accessible to VS-mode.

When hcounteren.HPM22 && scounteren.HPM22 are both set, hpmcounter22 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM23 Field

Location:

23

Description:

When set, the hpmcounter23 CSR (an alias of mhpmcounter23) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM23 is also set, hpmcounter23 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM23 is also set, hpmcounter23 is further accessible to VS-mode.

When hcounteren.HPM23 && scounteren.HPM23 are both set, hpmcounter23 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM24 Field

Location:

24

Description:

When set, the hpmcounter24 CSR (an alias of mhpmcounter24) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM24 is also set, hpmcounter24 is further accessible to U-mode.

<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM24 is also set, hpmcounter24 is further accessible to VS-mode.

When hcounteren.HPM24 && scounteren.HPM24 are both set, hpmcounter24 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM25 Field

Location:

25

Description:

When set, the hpmcounter25 CSR (an alias of mhpmcounter25) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM25 is also set, hpmcounter25 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM25 is also set, hpmcounter25 is further accessible to VS-mode.

When hcounteren.HPM25 && scounteren.HPM25 are both set, hpmcounter25 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM26 Field

Location:

26

Description:

When set, the hpmcounter26 CSR (an alias of mhpmcounter26) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM26 is also set, hpmcounter26 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM26 is also set, hpmcounter26 is further accessible to VS-mode.

When hcounteren.HPM26 && scounteren.HPM26 are both set, hpmcounter26 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM27 Field

Location:

27

Description:

When set, the hpmcounter27 CSR (an alias of mhpmcounter27) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM27 is also set, hpmcounter27 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM27 is also set, hpmcounter27 is further accessible to VS-mode.

When hcounteren.HPM27 && scounteren.HPM27 are both set, hpmcounter27 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM28 Field

Location:

28

Description:

When set, the hpmcounter28 CSR (an alias of mhpmcounter28) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM28 is also set, hpmcounter28 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM28 is also set, hpmcounter28 is further accessible to VS-mode.

When hcounteren.HPM28 && scounteren.HPM28 are both set, hpmcounter28 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM29 Field

Location:

29

Description:

When set, the hpmcounter29 CSR (an alias of mhpmcounter29) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM29 is also set, hpmcounter29 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM29 is also set, hpmcounter29 is further accessible to VS-mode.

When hcounteren.HPM29 && scounteren.HPM29 are both set, hpmcounter29 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM30 Field

Location:

30

Description:

When set, the hpmcounter30 CSR (an alias of mhpmcounter30) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM30 is also set, hpmcounter30 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM30 is also set, hpmcounter30 is further accessible to VS-mode.

When hcounteren.HPM30 && scounteren.HPM30 are both set, hpmcounter30 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM31 Field

Location:

31

Description:

When set, the hpmcounter31 CSR (an alias of mhpmcounter31) is accessible to

<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM31 is also set, hpmcounter31 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM31 is also set, hpmcounter31 is further accessible to VS-mode.

When hcounteren.HPM31 && scounteren.HPM31 are both set, hpmcounter31 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.8. 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.8.1. Attributes

CSR Address	0x320

Defining extension	• anyOf:
	  • Sm, version >= Sm@1.11.0
	  • Smhpm, version >= Smhpm@1.11.0

Length	32-bit
Privilege Mode	M

C.8.2. Format

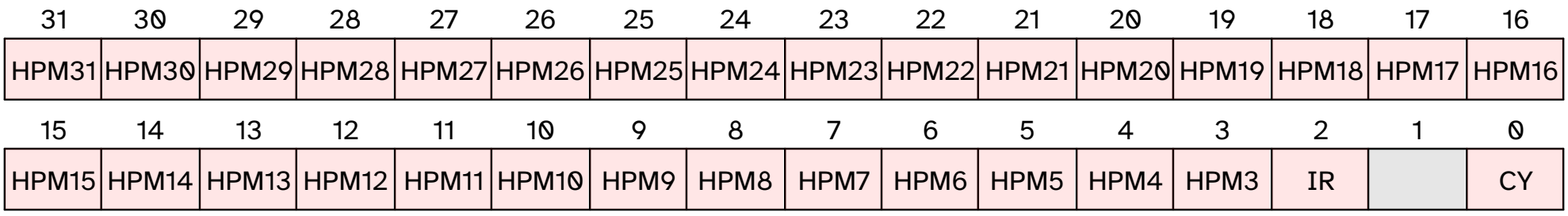


Figure 8. mcountinhibit format

C.8.3. 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
mcountinhibit.HPM8	8	RW RO	UNDEFINED_LEGAL

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

Name	Location	Type	Reset Value
mcountinhibit.HPM31	31	RW RO	UNDEFINED_LEGAL

C.8.4. 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.9. 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.9.1. Attributes

CSR Address	0xb00
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	M

C.9.2. Format

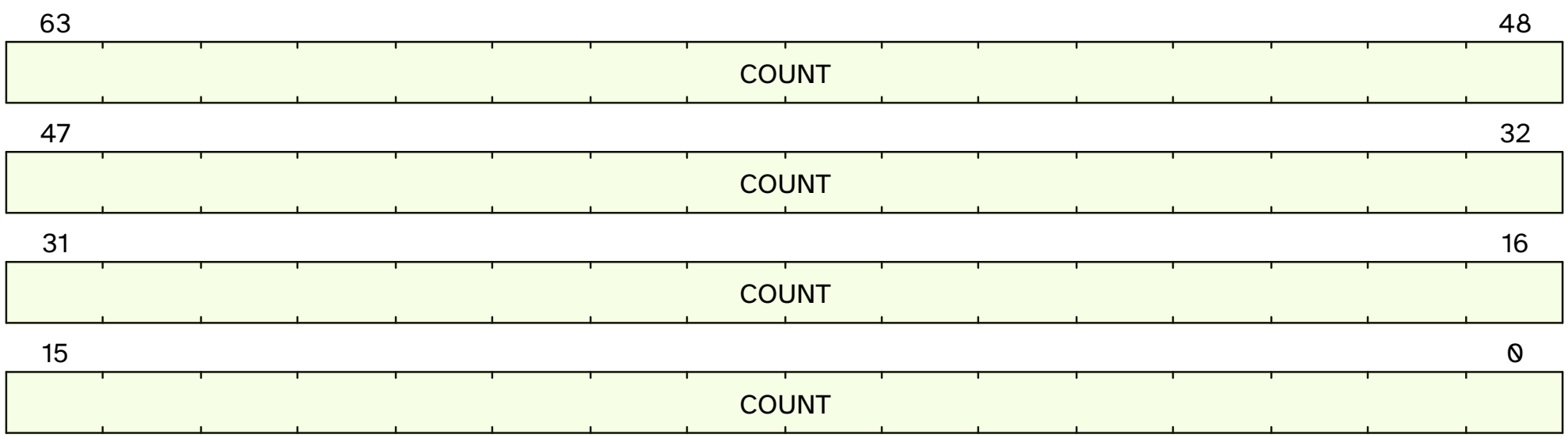


Figure 9. mcycle format

C.9.3. Field Summary

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

C.9.4. 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.9.5. 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.9.6. Software read

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

```
return read_mcycle();
```

C.10. mcycleh

High-half machine Cycle Counter

mcycleh is only defined in RV32.

High-half alias of mcycle.

C.10.1. Attributes

CSR Address	0xb80
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	M

C.10.2. Format

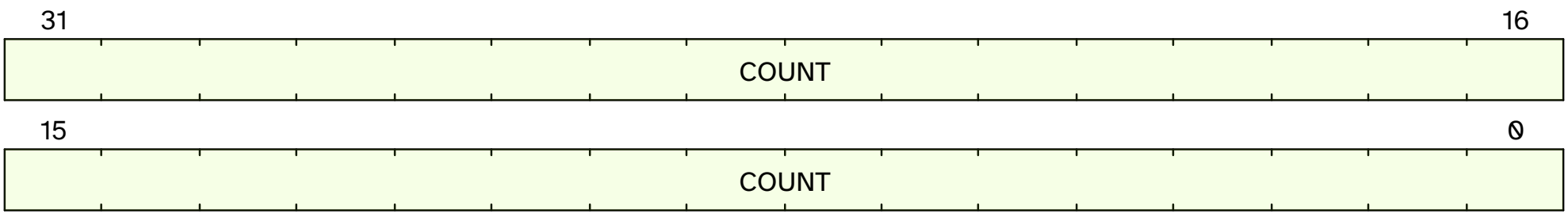


Figure 10. mcycleh format

C.10.3. Field Summary

Name	Location	Type	Reset Value
mcycleh.COUNT	31:0	RW-RH	UNDEFINED_LEGAL

C.10.4. Fields

mcycleh.COUNT Field

Location:

31:0

Description:

Alias of upper half of mcycle.COUNT.

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

C.10.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
  return sw_write_mcycle({csr_value.COUNT[31:0], read_mcycle()[31:0]});
} else {
  return sw_write_mcycle(csr_value.COUNT);
}
```

C.10.6. Software read

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

```
return read_mcycle()[63:32];
```

C.11. medeleg

Machine Exception Delegation

Controls exception delegation from M-mode to (H)S-mode <%- if ext?(:H) -%> or, in conjunction with hedeleg, to VS-mode <%- end -%> .

An exception cause is delegated to (H)S-mode when all of the following hold:

- The corresponding field in medeleg is set.
- The current privilege level is not M-mode. <%- if ext?(:H) -%>
- The same field in hedeleg is clear. <%- end -%>

<%- if ext?(:H) -%> An exception cause is delegated to VS-mode when all of the following hold:

- The corresponding field in medeleg is set.
- The corresponding field in hedeleg is set.
- The current privilege level is not M-mode or HS-mode. <%- end -%>

Otherwise, an exception cause is handled by M-mode.

See interrupt documentation for more details.

C.11.1. Attributes

CSR Address	0x302
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	M

C.11.2. Format

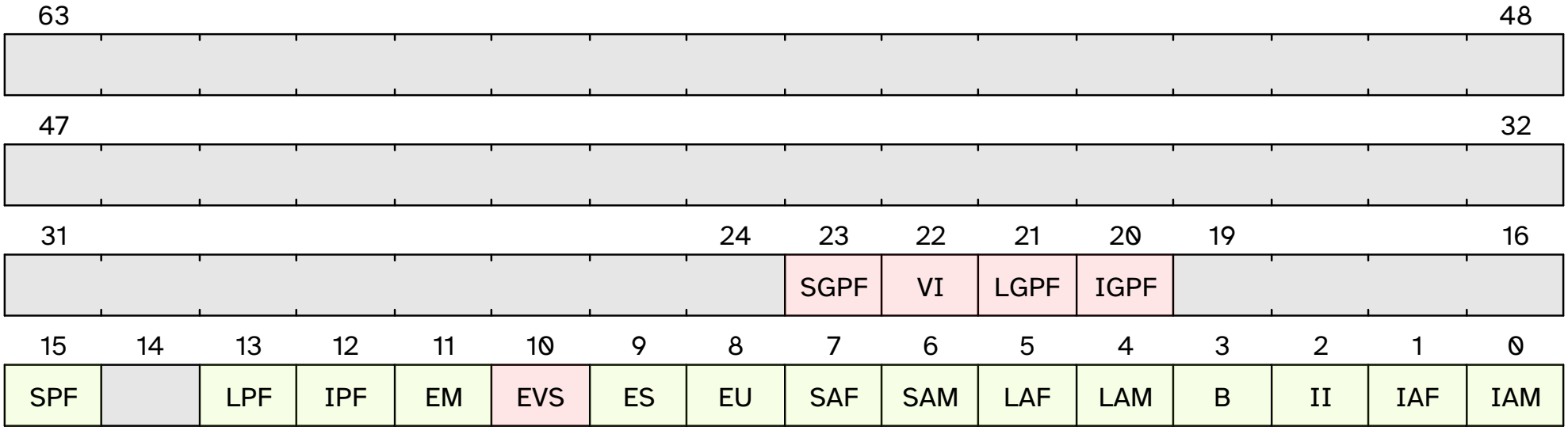


Figure 11. medeleg format

C.11.3. Field Summary

Name	Location	Type	Reset Value
medeleg.IAM	0	RW	UNDEFINED_LEGAL
medeleg.IAF	1	RW	UNDEFINED_LEGAL
medeleg.II	2	RW	UNDEFINED_LEGAL
medeleg.B	3	RW	UNDEFINED_LEGAL
medeleg.LAM	4	RW	UNDEFINED_LEGAL
medeleg.LAF	5	RW	UNDEFINED_LEGAL
medeleg.SAM	6	RW	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
medeleg.SAF	7	RW	UNDEFINED_LEGAL
medeleg.EU	8	RW	UNDEFINED_LEGAL
medeleg.ES	9	RW	UNDEFINED_LEGAL
medeleg.EVS	10	RW	UNDEFINED_LEGAL
medeleg.EM	11	RO	0
medeleg.IPF	12	RW	UNDEFINED_LEGAL
medeleg.LPF	13	RW	UNDEFINED_LEGAL
medeleg.SPF	15	RW	UNDEFINED_LEGAL
medeleg.IGPF	20	RW	UNDEFINED_LEGAL
medeleg.LGPF	21	RW	UNDEFINED_LEGAL
medeleg.VI	22	RW	UNDEFINED_LEGAL
medeleg.SGPF	23	RW	UNDEFINED_LEGAL

C.11.4. Fields

medeleg.IAM Field

Location:

0

Description:

Instruction Address Misaligned

Delegates Instruction Address Misaligned exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Address Misaligned exceptions may be further delegated to VS-mode if hedeleg.IAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAM .2+! hedeleg.IAM 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAM 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.IAF Field

Location:

1

Description:

Instruction Access Fault

Delegates Instruction Access Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Access Fault exceptions may be further delegated to VS-mode if hedeleg.IAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAF .2+! hedeleg.IAF 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.II Field

Location:

2

Description:

Illegal Instruction

Delegates Illegal Instruction exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Illegal Instruction exceptions may be further delegated to VS-mode if hedeleg.II is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.II .2+! hedeleg.II 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.II 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.B Field

Location:

3

Description:

Breakpoint

Delegates Breakpoint exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Breakpoint exceptions may be further delegated to VS-mode if hedeleg.B is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.B .2+! hedeleg.B 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.B 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LAM Field

Location:

4

Description:

Load Address Misaligned

Delegates Load Address Misaligned exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Address Misaligned exceptions may be further delegated to VS-mode if hedeleg.LAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

[when,"MISALIGNED_LDST == true"]
Note that because this implementation supports misaligned loads, this exception will never occur.
However, the writable bit should be presented anyway.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAM .2+! hedeleg.LAM 4+^.>! Current Mode
.>h! `M-mode` .>h! `HS-mode` .>h! `VS-mode` .>h! `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAM 2+^.>! Current Mode
.>h! `M-mode` .>h! `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LAF Field

Location:

5

Description:

Load Access Fault

Delegates Load Access Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Access Fault exceptions may be further delegated to VS-mode if hedeleg.LAF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

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

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAF .2+! hedeleg.LAF 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SAM Field

Location:

6

Description:

Store/AMO Address Misaligned

Delegates Store/AMO Address Misaligned exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Address Misaligned exceptions may be further delegated to VS-mode if hedeleg.SAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

[when,"MISALIGNED_LDST == true && MISALIGNED_AMO == true"]
Note that because the implementation supports misaligned stores and misaligned AMOs (or no AMOs), this exception will never occur.
Even so, the writable bit should be presented anyway.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAM .2+! hedeleg.SAM 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAM 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SAF Field

Location:

7

Description:

Store/AMO Access Fault

Delegates Store/AMO Access Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Access Fault exceptions may be further delegated to VS-mode if hedeleg.SAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAF .2+! hedeleg.SAF 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.EU Field

Location:

8

Description:

Environment Call from U-Mode

Delegates Environment Call from U-mode exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Environment Call from U-mode exceptions may be further delegated to VS-mode if hedeleg.EU is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.EU .2+! hedeleg.EU 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.EU 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.ES Field

Location:

9

Description:

Environment Call from S-Mode

Delegates Environment Call from S-mode exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Environment Call from S-mode exceptions cannot be delegated to VS-mode.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.ES 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! (H)S
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.EVS Field

Location:

10

Description:

Environment Call from VS-Mode

Delegates Environment Call from VS-mode exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Environment Call from S-mode exceptions cannot be delegated to VS-mode.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.EVS 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! (H)S
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.EM Field

Location:

11

Description:

Environment Call from M-Mode

An Environment Call from M-mode cannot be delegated, so this is a read-only field.

All Environment Call from M-mode exceptions are taken by M-mode.

Type:

RO

Reset value:

0

medeleg.IPF Field

Location:

12

Description:

Instruction Page Fault

Delegates Instruction Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Page Fault exceptions may be further delegated to VS-mode if hedeleg.IPF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IPF .2+! hedeleg.IPF 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IPF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M

! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LPF Field

Location:

13

Description:

Load Page Fault

Delegates Load Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Page Fault exceptions may be further delegated to VS-mode if hedeleg.LPF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LPF .2+! hedeleg.LPF 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LPF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SPF Field

Location:

15

Description:

Store/AMO Page Fault

Delegates Store/AMO Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Page Fault exceptions may be further delegated to VS-mode if hedeleg.SPF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SPF .2+! hedeleg.SPF 4+^.>! Current Mode
.>h! `M-mode` *.>h!* `HS-mode` *.>h!* `VS-mode` *.>h!* `VU-mode`

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LPF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `S-mode`

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.IGPF Field

Location:

20

Description:

Instruction Guest Page Fault

Delegates Instruction Guest Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Guest Page Fault exceptions cannot be delegated to VS-mode.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IGPF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LGPF Field

Location:

21

Description:

Load Guest Page Fault

Delegates Load Guest Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Guest Page Fault exceptions cannot be delegated to VS-mode.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LGPF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.VI Field

Location:

22

Description:

Virtual Instruction

Delegates Virtual Instruction exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Virtual Instruction exceptions cannot be delegated to VS-mode.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.VI 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SGPF Field

Location:

23

Description:

Store/AMO Guest Page Fault

Delegates Store/AMO Guest Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Guest Page Fault exceptions cannot be delegated to VS-mode.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SGPF 2+^.>! Current Mode
.>h! `M-mode` *.>h!* `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

C.12. menvcfg

Machine Environment Configuration

Contains fields that control certain characteristics of the execution environment for modes less privileged than M-mode.

The menvcfg CSR controls certain characteristics of the execution environment for modes less privileged than M.

If bit FIOM (Fence of I/O implies Memory) is set to one in menvcfg, FENCE instructions executed in modes less privileged than M are modified so the requirement to order accesses to device I/O implies also the requirement to order main memory accesses. Table 11 details the modified interpretation of FENCE instruction bits PI, PO, SI, and SO for modes less privileged than M when FIOM=1.

Similarly, for modes less privileged than M when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its *aq* and/or *rl* bit set, then that instruction is ordered as though it accesses both device I/O and memory.

If S-mode is not supported, or if satp.MODE is read-only zero (always Bare), the implementation may make FIOM read-only zero.

Table 11. Modified interpretation of FENCE predecessor and successor sets for modes less privileged than M when FIOM=1.

Instruction bit |Meaning when set	PI PO
Predecessor device input and memory reads (PR implied) Predecessor device output and memory writes (PW implied)	SI SO

Bit FIOM is needed in menvcfg so M-mode can emulate the H (hypervisor) extension, which has an equivalent FIOM bit in the hypervisor CSR henvcfg.

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode and G-stage address translation (i.e., for page tables pointed to by satp or hgatp). When PBMTE=1, Svpbmt is available for S-mode and G-stage address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read-only zero. Furthermore, for implementations with the hypervisor extension, henvcfg.PBMTE is read-only zero if menvcfg.PBMTE is zero.

After changing menvcfg.PBMTE, executing an SFENCE.VMA instruction with *rs1*=`x0` and *rs2*=`x0` suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' PBMT fields. See [hyp-mm-fences] for additional synchronization requirements when the hypervisor extension is implemented.

If the Svadu extension is implemented, the ADUE bit controls whether hardware updating of PTE A/D bits is enabled for S-mode and G-stage address translations. When ADUE=1, hardware updating of PTE A/D bits is enabled during S-mode address translation, and the implementation behaves as though the Svade extension were not implemented for S-mode address translation. When the hypervisor extension is implemented, if ADUE=1, hardware updating of PTE A/D bits is enabled during G-stage address translation, and the implementation behaves as though the Svade extension were not implemented for G-stage address translation. When ADUE=0, the implementation behaves as though Svade were implemented for S-mode and G-stage address translation. If Svadu is not implemented, ADUE is read-only zero. Furthermore, for implementations with the hypervisor extension, henvcfg.ADUE is read-only zero if menvcfg.ADUE is zero.

The Svade extension requires page-fault exceptions be raised when PTE A/D bits need be set, hence Svade is implemented when ADUE=0.

If the Smcdeleg extension is implemented, the CDE (Counter Delegation Enable) bit controls whether Zicntr and Zihpm counters can be delegated to S-mode. When CDE=1, the Smcdeleg extension is enabled, see [smcdeleg]. When CDE=0, the Smcdeleg and Ssccfg extensions appear to be not implemented. If Smcdeleg is not implemented, CDE is read-only zero.

The definition of the STCE field is furnished by the Sstc extension.

The definition of the CBZE field is furnished by the Zicboz extension.

The definitions of the CBCFE and CBIE fields are furnished by the Zicbom extension.

The definition of the PMM field will be furnished by the forthcoming Smnpm extension. Its allocation within menvcfg may change prior to the ratification of that extension.

The Zicfilp extension adds the `LPE` field in menvcfg. When the `LPE` field is set to 1 and S-mode is implemented, the Zicfilp extension is enabled in S-mode. If `LPE` field is set to 1 and S-mode is not implemented, the Zicfilp extension is enabled in U-mode. When the `LPE` field is 0, the Zicfilp extension is not enabled in S-mode, and the following rules apply to S-mode. If the `LPE` field is 0 and S-mode is not implemented, then the same rules apply to U-mode.

- The hart does not update the `ELP` state; it remains as `NO_LP_EXPECTED`.
- The `LPAD` instruction operates as a no-op.

The Zicfiss extension adds the `SSE` field to menvcfg. When the `SSE` field is set to 1 the Zicfiss extension is activated in S-mode. When `SSE` field is 0, the following rules apply to privilege modes that are less than M:

- 32-bit Zicfiss instructions will revert to their behavior as defined by Zimop.
- 16-bit Zicfiss instructions will revert to their behavior as defined by Zcmop.
- The `pte.xwr=010b` encoding in VS/S-stage page tables becomes reserved.
- The henvcfg and senvcfg fields will read as zero and are read-only.
- `SSAMOSWAP.W/D` raises an illegal-instruction exception.

The Ssdbltrp extension adds the double-trap-enable (`DTE`) field in menvcfg. When menvcfg is zero, the implementation behaves as though Ssdbltrp is not

implemented. When Ssdbltrp is not implemented sstatus, vsstatus, and henvcfg bits are read-only zero.

When XLEN=32, menvcfgh is a 32-bit read/write register that aliases bits 63:32 of menvcfg. The menvcfgh register does not exist when XLEN=64.

If U-mode is not supported, then registers menvcfg and menvcfgh do not exist.

C.12.1. Attributes

CSR Address	0x30a

Defining extension	• allOf:
	  • Sm, version >=1.12
	  • U, version >= U@1.0.0

Length	64-bit
Privilege Mode	M

C.12.2. Format

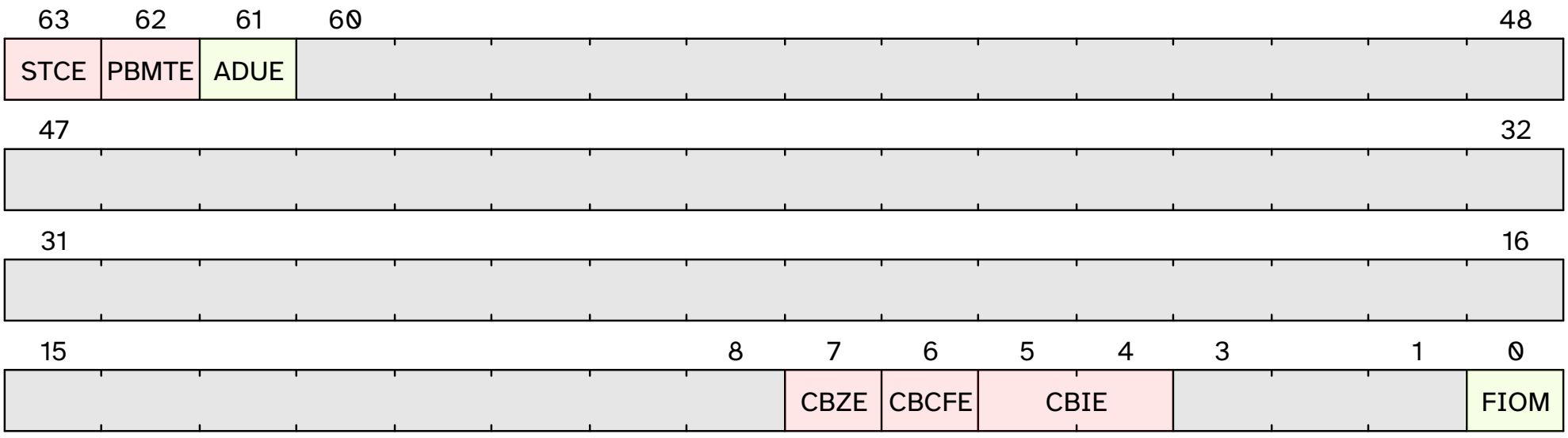


Figure 12. menvcfg format

C.12.3. Field Summary

Name	Location	Type	Reset Value
menvcfg.STCE	63	RW	UNDEFINED_LEGAL
menvcfg.PBMTE	62	RW	UNDEFINED_LEGAL
menvcfg.CBZE	7	RW	UNDEFINED_LEGAL
menvcfg.CBCFE	6	RW	UNDEFINED_LEGAL
menvcfg.CBIE	5:4	RW-R	UNDEFINED_LEGAL
menvcfg.FIOM	0	RW	UNDEFINED_LEGAL

C.12.4. Fields

menvcfg.STCE Field

Location:

63

Description:

STimecmp Enable

When set, `stimecmp` *is operational.*

When clear, `stimecmp` *access in a mode other than M-mode raises an* `Illegal Instruction` *trap.*

S-mode timer interrupts will not be generated when clear, and mip and sip revert to their prior behavior without Sstc.

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.PBMTE Field

Location:

62

Description:

Page Based Memory Type Enable

The PBMTE bit controls whether the Svpbmt extension is available for use in S-mode<% if ext?(:H) %>and G-stage<% end %> address translation (i.e., for page tables pointed to by satp<% if ext?(:H) %> or hgatp<% end %>). When PBMTE=1, Svpbmt is available for S-mode <% if ext?(:H) %> and G-stage <% end %> address translation. When PBMTE=0, the implementation behaves as though Svpbmt were not implemented. If Svpbmt is not implemented, PBMTE is read-only zero.

<% if ext?(:H) %>
Furthermore, henvcfg.PBMTE is read-only zero if
menvcfg.PBMTE is zero.
<% end %>

After changing menvcfg.PBMTE, executing an sfence.vma instruction with rs1=x0 and rs2=x0 suffices to synchronize address-translation caches with respect to the altered interpretation of page-table entries' PBMT fields.

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.CBZE Field

Location:

7

Description:

Cache Block Zero instruction Enable

Enables the execution of the cache block zero instruction, `CBO.ZERO`*,*
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- `0`*: The instruction raises an illegal instruction or virtual instruction exception*
- `1`*: The instruction is executed*

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.CBCFE Field

Location:

6

Description:

Cache Block Clean and Flush instruction Enable

Enables the execution of the cache block clean instruction, `CBO.CLEAN`, and the cache block flush instruction, `CBO.FLUSH`,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- `0`: *The instruction raises an illegal instruction or virtual instruction exception*
- `1`: *The instruction is executed*

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfg.CBIE Field

Location:

5:4

Description:

Cache Block Invalidate instruction Enable

Enables the execution of the cache block invalidate instruction, `CBO.INVAL`,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- `00`: *The instruction raises an illegal instruction or virtual instruction exception*
- `01`: *The instruction is executed and performs a flush operation*
- `10`: *Reserved*
- `11`: *The instruction is executed and performs an invalidate operation*

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

menvcfg.FIOM Field

Location:

`0`

Description:

Fence of I/O implies Memory

When menvcfg.FIOM is set,
FENCE instructions ordering I/O regions also implicitly order memory regions when executed in any mode less privileged than M-mode.

[separator="!",%autowidth,float="center",align="center",cols="^,<",options="header"]
!===
!Instruction bit !Meaning when set
!PI +
PO
!Predecessor device input and memory reads (PR implied) +
Predecessor device output and memory writes (PW implied)
!SI +
SO
!Successor device input and memory reads (SR implied) +
Successor device output and memory writes (SW implied)
!===

Similarly, for modes less privileged than M when FIOM=1, if an atomic instruction that accesses a region ordered as device I/O has its aq and/or rl bit set, then that instruction is ordered as though it accesses both device I/O and memory.

Type:

RW

Reset value:

UNDEFINED_LEGAL

C.12.5. 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:

```
STCE = csr_value.STCE
PBMTE = csr_value.PBMTE
CBZE = csr_value.CBZE
CBCFE = csr_value.CBCFE
CBIE = if ((csr_value.CBIE == 0) ||
    (csr_value.CBIE == 1) ||
    ((!FORCE_UPGRADE_CBO_INVAL_TO_FLUSH) && (csr_value.CBIE == 3))) {
  return csr_value.CBIE;
} else {
  return CSR[menvcfg].CBIE;
}

FIOM = csr_value.FIOM
```

C.13. menvcfgh

Machine Environment Configuration

menvcfgh is only defined in RV32.

Contains bits to enable/disable extensions

C.13.1. Attributes

CSR Address	0x31a

Defining extension	• allOf:
	  • Sm, version >=1.12
	  • U, version >= U@1.0.0

Length	32-bit
Privilege Mode	M

C.13.2. Format

Figure 13. menvcfgh format

C.13.3. Field Summary

Name	Location	Type	Reset Value
menvcfgh.STCE	31	RW	UNDEFINED_LEGAL
menvcfgh.PBMTE	30	RW	UNDEFINED_LEGAL

C.13.4. Fields

menvcfgh.STCE Field

Location:

31

Description:

STimecmp Enable

Alias of menvcfg.STCE

Type:

RW

Reset value:

UNDEFINED_LEGAL

menvcfgh.PBMTE Field

Location:

30

Description:

Page Based Memory Type Enable

Alias of menvcfg.PBMTE

Type:

RW

Reset value:

UNDEFINED_LEGAL

C.14. 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.14.1. Attributes

CSR Address	0x341
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.14.2. Format

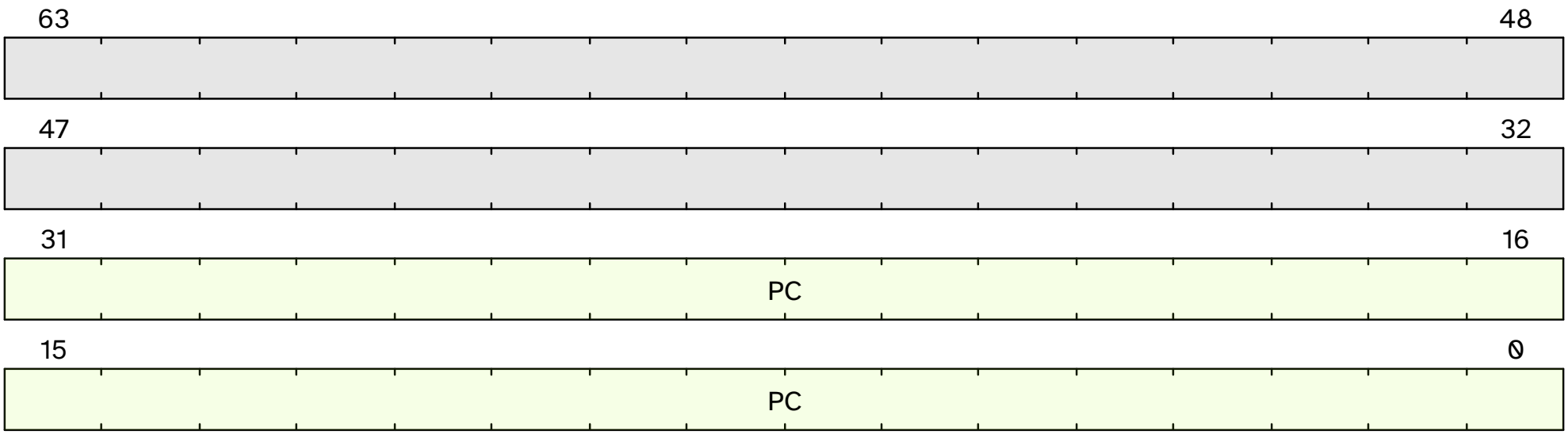


Figure 14. mepc format

C.14.3. Field Summary

Name	Location	Type	Reset Value
mepc.PC	63:0	RW-RH	0

C.14.4. Fields

mepc.PC Field

Location:

63:0

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.14.5. 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.14.6. Software read

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

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

C.15. mhartid

Machine Hart ID

Reports the unique hart-specific ID in the system.

C.15.1. Attributes

CSR Address	0xf14
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.15.2. Format

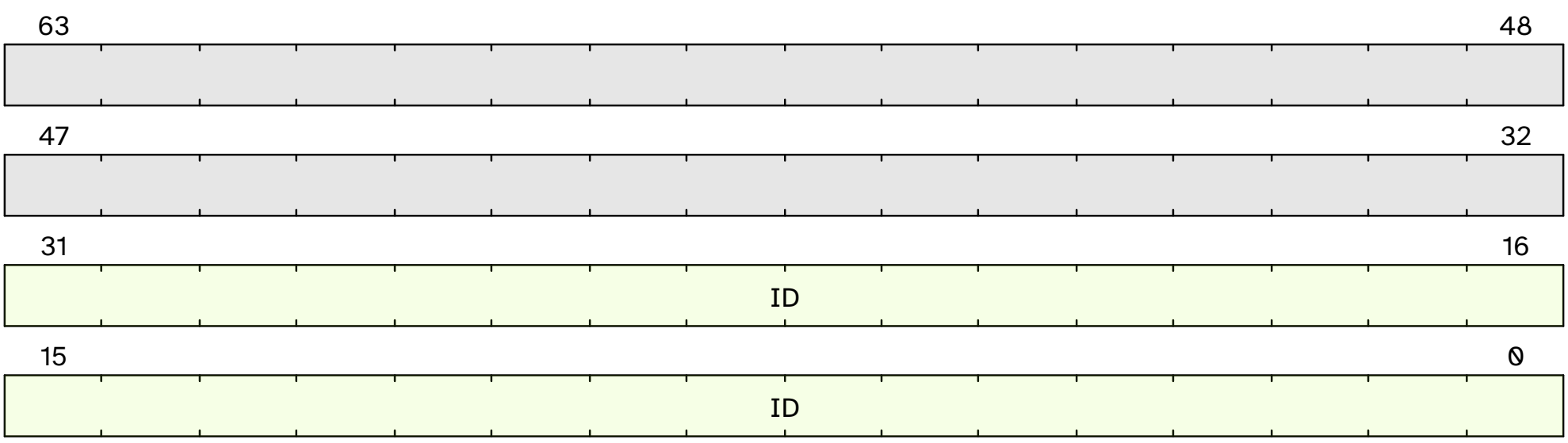


Figure 15. mhartid format

C.15.3. Field Summary

Name	Location	Type	Reset Value
mhartid.ID	63:0	RO	UNDEFINED_LEGAL

C.15.4. Fields

mhartid.ID Field

Location:

63:0

Description:

hart-specific ID.

Type:

RO

Reset value:

UNDEFINED_LEGAL

C.15.5. Software read

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

```
return hartid();
```

C.16. mideleg

Machine Interrupt Delegation

Controls exception delegation from M-mode to HS/S-mode

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the `MRET` instruction. To increase performance, implementations can provide individual read/write bits within mideleg to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level.

In harts with S-mode, the mideleg register must exist, and setting a bit mideleg will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler <%- if ext?(:H) -%> (which could further be delegated to VS-mode through hideleg) <%- end -%> . <%- if ext?(:S, ">
1.9.1") -%> In harts without S-mode, the mideleg register should not exist.

In versions 1.9.1 and earlier , this register existed but was hardwired to zero in M-mode only, or M/U without N harts. There is no reason to require they return zero in those cases, as the misa register indicates whether they exist.

<%- else -%> In harts without S-mode, the mideleg register is read-only zero. <%- end -%>

An implementation can choose to subset the delegatable traps, with the supported delegatable bits found by writing one to every bit location, then reading back the value in mideleg to see which bit positions hold a one.

Version 1.11 and earlier prohibited having any bits of mideleg be read-only one. Platform standards may always add such restrictions.

Traps never transition from a more-privileged mode to a less-privileged mode. For example, if M-mode has delegated illegal-instruction exceptions to S-mode, and M-mode software later executes an illegal instruction, the trap is taken in M-mode, rather than being delegated to S-mode. By contrast, traps may be taken horizontally. Using the same example, if M-mode has delegated illegal-instruction exceptions to S-mode, and S-mode software later executes an illegal instruction, the trap is taken in S-mode.

Delegated interrupts result in the interrupt being masked at the delegator privilege level. For example, if the supervisor timer interrupt (STI) is delegated to S-mode by setting mideleg[5], STIs will not be taken when executing in M-mode. By contrast, if mideleg[5] is clear, STIs can be taken in any mode and regardless of current mode will transfer control to M-mode.

mideleg holds trap delegation bits for individual interrupts, with the layout of bits matching those in the mip register (i.e., STIP interrupt delegation control is located in bit 5).

For exceptions that cannot occur in less privileged modes, the corresponding medeleg bits should be read-only zero. In particular, medeleg[11] is read-only zero.

The medeleg[16] is read-only zero as double trap is not delegatable.

C.16.1. Attributes

CSR Address	0x303

Defining extension	• oneOf:
	  • Sm, version ⇐ 1.9.1
	  • allOf:
	  ■ S, version > 1.9.1
	  ■ Sm, version > 1.9.1

Length	64-bit
Privilege Mode	M

C.16.2. Format

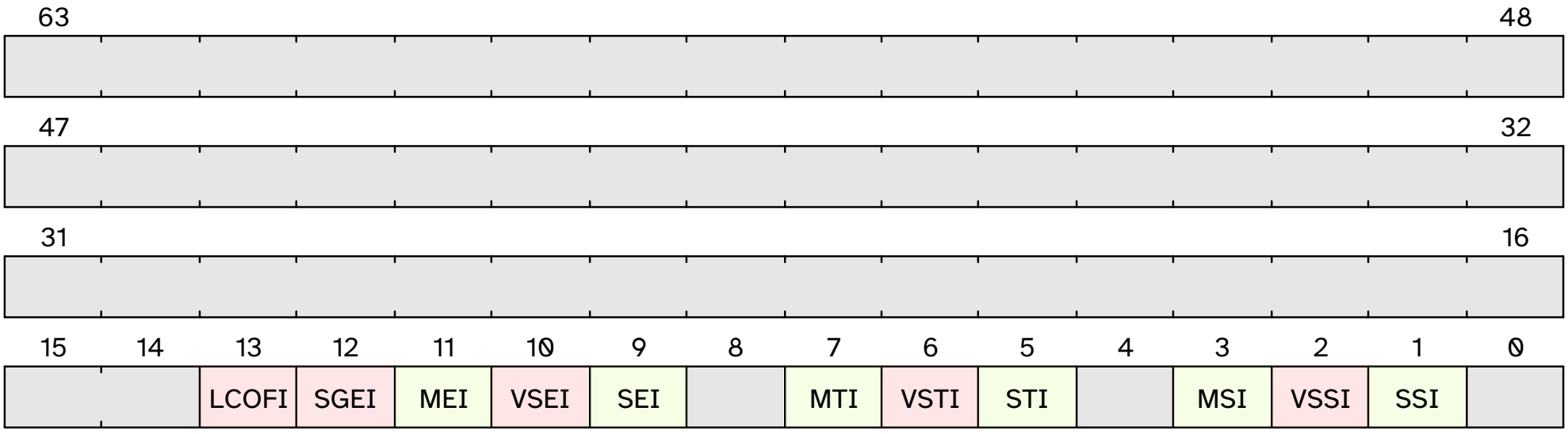


Figure 16. mideleg format

C.16.3. Field Summary

Name	Location	Type	Reset Value
mideleg.SSI	1	RW	0
mideleg.VSSI	2	RO	1
mideleg.MSI	3	RO	0
mideleg.STI	5	RW	0
mideleg.VSTI	6	RO	1
mideleg.MTI	7	RO	0
mideleg.SEI	9	RW	0
mideleg.VSEI	10	RO	1
mideleg.MEI	11	RO	0
mideleg.SGEI	12	RO	1
mideleg.LCOFI	13	RW	0

C.16.4. Fields

mideleg.SSI Field

Location:

1

Description:

Supervisor Software Interrupt delegation

When 1, Supervisor Software interrupts are delegated to HS/S-mode.

Type:

RW

Reset value:

0

mideleg.VSSI Field

Location:

2

Description:

Virtual Supervisor Software Interrupt delegation

When 1, Virtual Supervisor Software interrupts are delegated to HS-mode.

Virtual Supervisor Software Interrupts are always delegated to HS-mode, so this field is read-only one.

Type:

RO

Reset value:

1

mideleg.MSI Field

Location:

3

Description:

Machine Software interrupt delegation

Since M-mode interrupts cannot be delegated, this field is read-only zero.

Type:

RO

Reset value:

0

mideleg.STI Field

Location:

5

Description:

Supervisor Timer interrupt delegation

When 1, Supervisor Timer interrupts are delegated to HS/S-mode.

Type:

RW

Reset value:

0

mideleg.VSTI Field

Location:

6

Description:

Virtual Supervisor Timer interrupt delegation

When 1, Virtual Supervisor Timer interrupts are delegated to HS-mode.

Virtual Supervisor Time Interrupts are always delegated to HS-mode, so this field is read-only one.

Type:

RO

Reset value:

1

mideleg.MTI Field

Location:

7

Description:

Machine Timer interrupt delegation

Since M-mode interrupts cannot be delegated, this field is read-only zero.

Type:

RO

Reset value:

0

mideleg.SEI Field

Location:

9

Description:

Supervisor External interrupt delegation

When 1, Supervisor External interrupts are delegated to HS/S-mode.

Type:

RW

Reset value:

0

mideleg.VSEI Field

Location:

10

Description:

Virtual Supervisor External interrupt delegation

Virtual Supervisor External Interrupts are always delegated to HS-mode, so this field is read-only one.

Type:

RO

Reset value:

1

mideleg.MEI Field

Location:

11

Description:

Machine External interrupt delegation

Since M-mode interrupts cannot be delegated, this field is read-only zero.

Type:

RO

Reset value:

0

mideleg.SGEI Field

Location:

12

Description:

Supervisor Guest External Interrupt delegation

Supervisor Guest External interrupts are always delegated to HS-mode, so this field is read-only one.

Type:

RO

Reset value:

1

mideleg.LCOFI Field

Location:

13

Description:

Local Counter Overflow Interrupt delegation

When 1, local counter overflow interrupts are delegated to (H)S-mode.

Type:

RW

Reset value:

0

C.17. mie

Machine Interrupt Enable

mip.yaml#/description

C.17.1. Attributes

CSR Address	0x304
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.17.2. Format

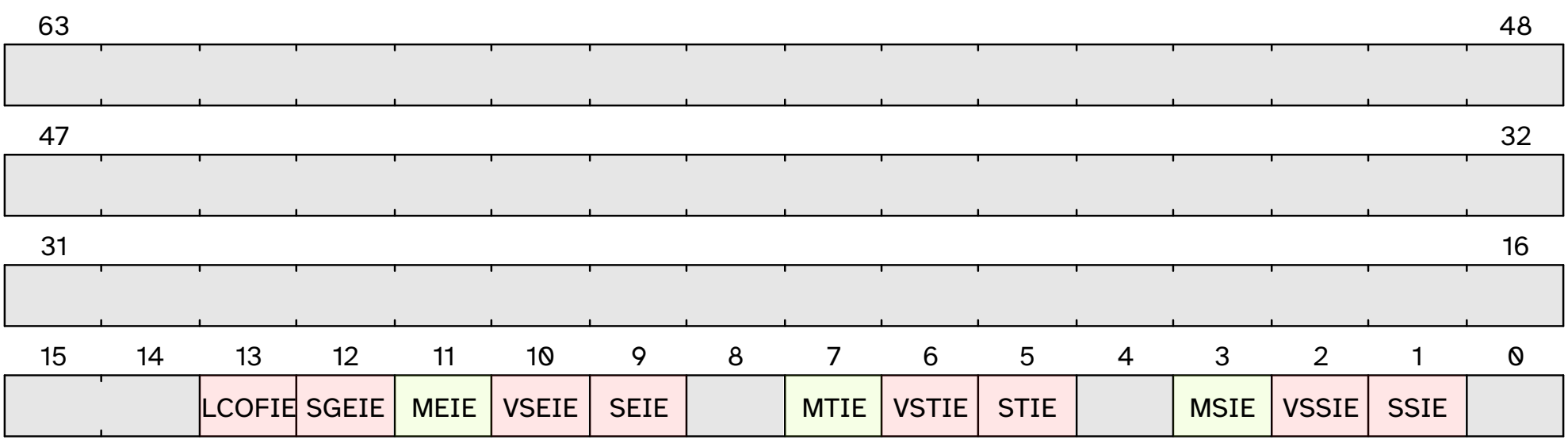


Figure 17. mie format

C.17.3. 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.17.4. 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.18. 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.18.1. Attributes

CSR Address	0xf13
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.18.2. Format

Figure 18. mimpid format

C.18.3. Field Summary

Name	Location	Type	Reset Value
mimpid.Implementation	63:0	RO	UNDEFINED_LEGAL

C.18.4. Fields

mimpid.Implementation Field

Location:

63:0

Description:

Vendor-specific implementation ID.

Type:

RO

Reset value:

UNDEFINED_LEGAL

C.19. 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.19.1. Attributes

CSR Address	0xb02
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	M

C.19.2. Format

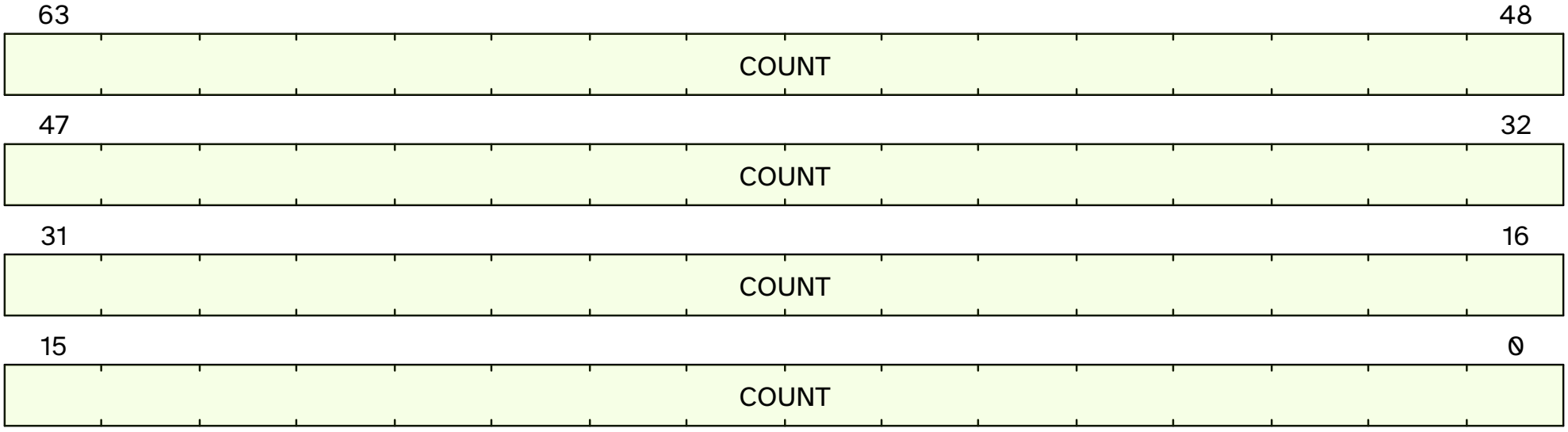


Figure 19. minstret format

C.19.3. Field Summary

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

C.19.4. 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.20. minstreth

Machine Instructions Retired Counter

minstreth is only defined in RV32.

Upper half of 64-bit instructions retired counters.

See minstret for details.

C.20.1. Attributes

CSR Address	0xb82
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	M

C.20.2. Format

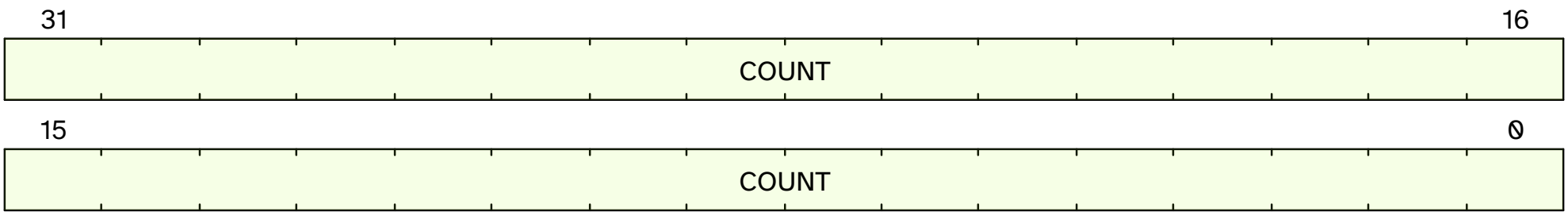


Figure 20. minstreth format

C.20.3. Field Summary

Name	Location	Type	Reset Value
minstreth.COUNT	31:0	RW-H	UNDEFINED_LEGAL

C.20.4. Fields

minstreth.COUNT Field

Location:

31:0

Description:

Instructions retired counter

Upper half of minstret.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.20.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = CSR[mcycle].COUNT = {csr_value.COUNT[31:0], CSR[minstret].COUNT[31:0]};
return csr_value.COUNT;
```

C.20.6. Software read

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

```
return minstret.COUNT[63:32];
```

C.21. 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.21.1. Attributes

CSR Address	0x344
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.21.2. Format

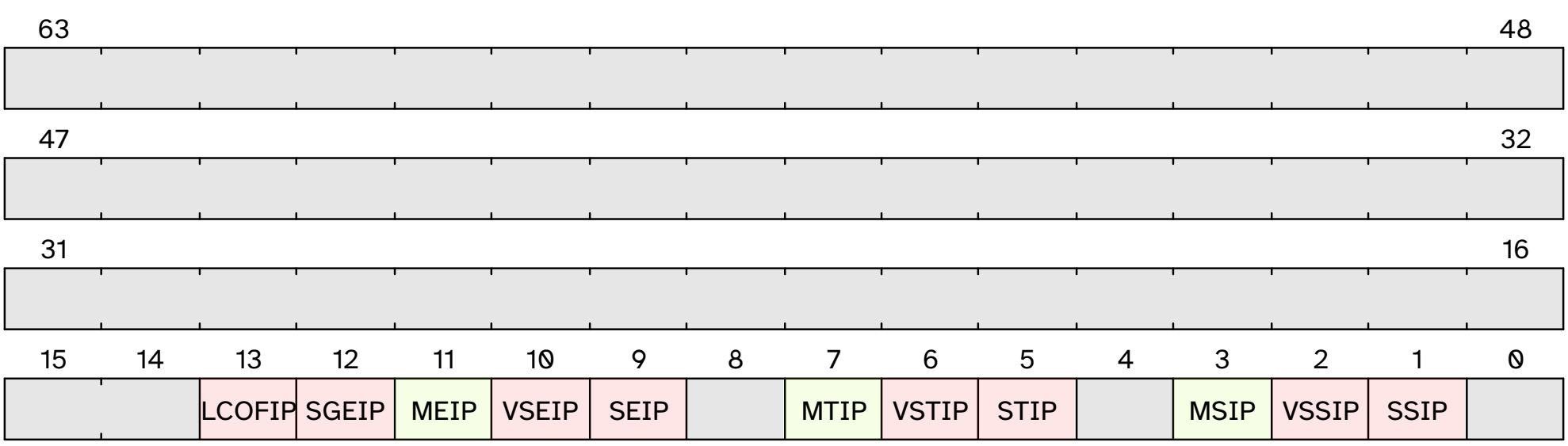


Figure 21. mip format

C.21.3. 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
mip.LCOFIP	13	RW-H	0

C.21.4. 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.21.5. Software read

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

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

C.22. misa

Machine ISA Control

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

C.22.1. Attributes

CSR Address	0x301
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.22.2. Format

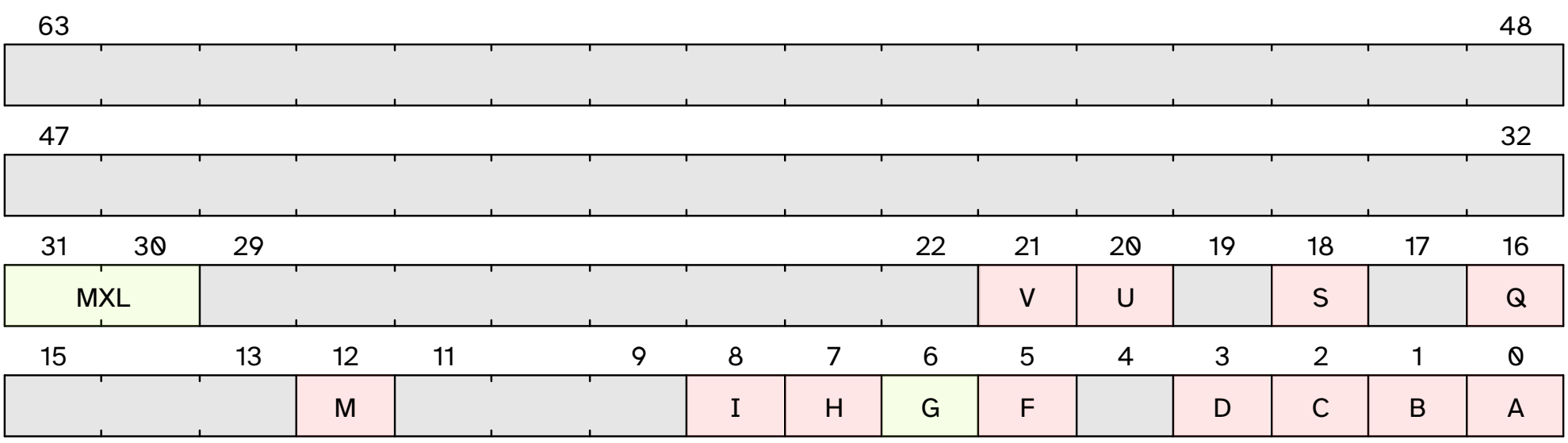


Figure 22. misa format

C.22.3. Field Summary

Name	Location	Type	Reset Value
misa.MXL	63:62	RO	2
misa.A	0	RW RO	UNDEFINED_LEGAL
misa.B	1	RW RO	1
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	1
misa.Q	16	RW RO	1
misa.S	18	RW RO	1

Name	Location	Type	Reset Value
misa.U	20	RW RO	1
misa.V	21	RW RO	UNDEFINED_LEGAL

C.22.4. Fields

misa.MXL Field

Location:

63:62

Description:

XLEN in M-mode.

Type:

RO

Reset value:

2

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:

1

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:

1

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:

1

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:

1

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.22.5. 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.22.6. Software read

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

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

C.23. mnepc

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.23.1. Attributes

CSR Address	0x741
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.23.2. Format

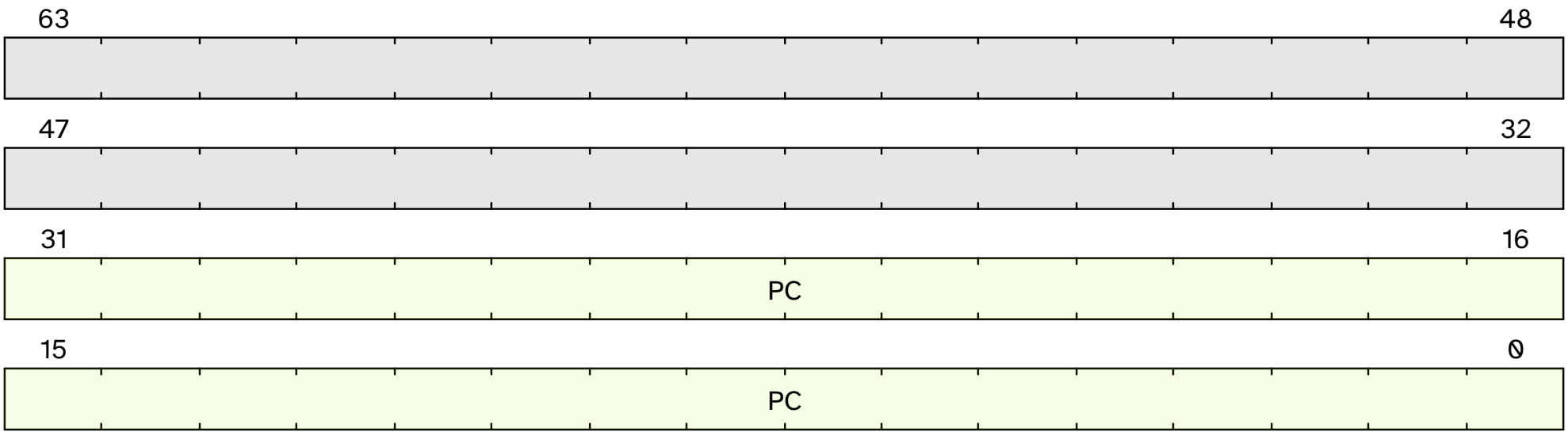


Figure 23. mnepc format

C.23.3. Field Summary

Name	Location	Type	Reset Value
mnepc.PC	63:0	RW-RH	UNDEFINED_LEGAL

C.23.4. Fields

mnepc.PC Field

Location:

63:0

Description:

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

On an exception return from M-mode NMI / double trap (from the MNRET instruction), control transfers to the virtual address read out of mnepc.PC.

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

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

UNDEFINED_LEGAL

C.23.5. 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.23.6. Software read

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

```
if (implemented?(ExtensionName::C) && misa.C == 1'b1) {
  return mnepc.PC & ~64'b1;
} else {
  return mnepc.PC & ~64'b11;
}
```

C.24. mscratch

Machine Scratch Register

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

C.24.1. Attributes

CSR Address	0x340
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.24.2. Format

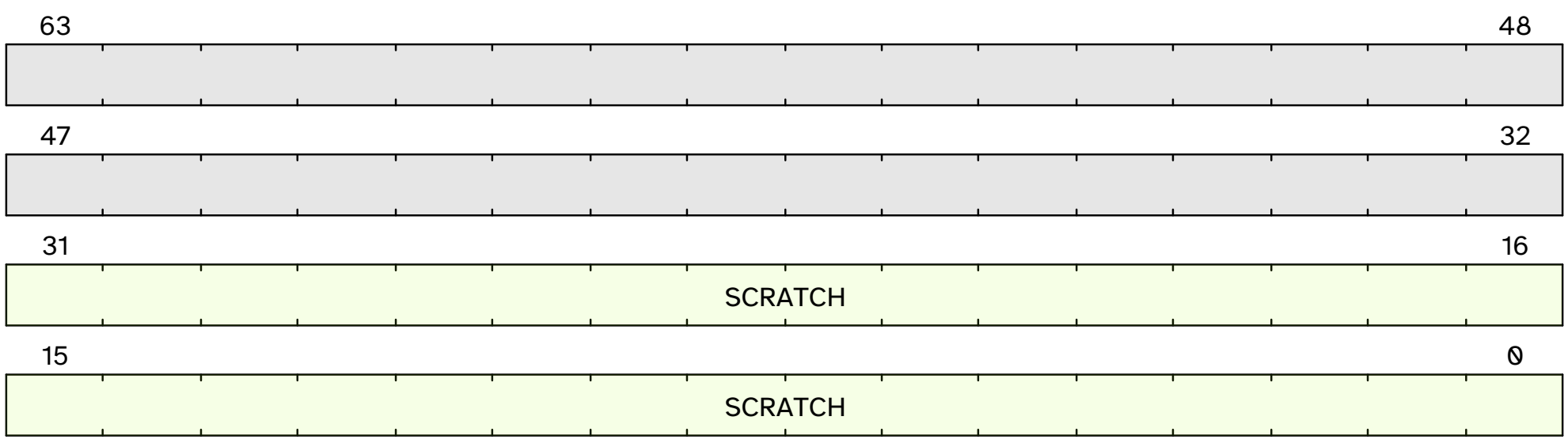


Figure 24. mscratch format

C.24.3. Field Summary

Name	Location	Type	Reset Value
mscratch.SCRATCH	63:0	RW	0

C.24.4. Fields

mscratch.SCRATCH Field

Location:

63:0

Description:

Scratch value

Type:

RW

Reset value:

0

C.25. mstatus

Machine Status

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

C.25.1. Attributes

CSR Address	0x300
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.25.2. Format

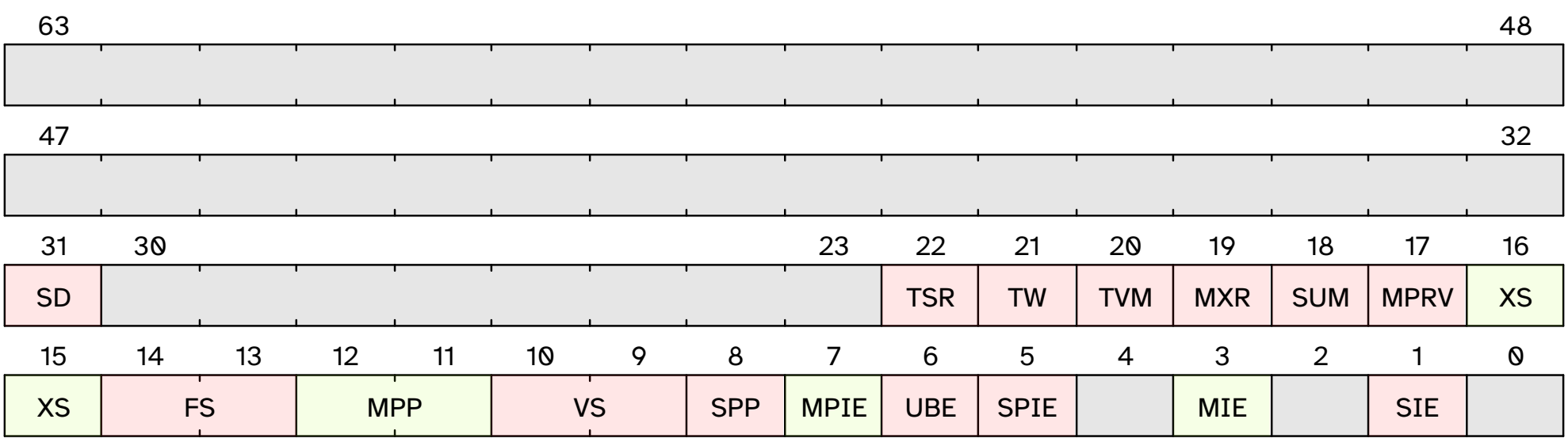


Figure 25. mstatus format

C.25.3. Field Summary

Name	Location	Type	Reset Value
mstatus.SD	63	RO-H RO-H RO	UNDEFINED_LEGAL
mstatus.MDT	42	RW-H	UNDEFINED_LEGAL
mstatus.MPV	39	RW-H	UNDEFINED_LEGAL
mstatus.GVA	38	RW-H	0
mstatus.MBE	37	RO	0
mstatus.SBE	36	RW RO	UNDEFINED_LEGAL
mstatus.SXL	35:34	RW RO	UNDEFINED_LEGAL
mstatus.UXL	33:32	RW RO	UNDEFINED_LEGAL
mstatus.TSR	22	RW	UNDEFINED_LEGAL
mstatus.TW	21	RW	UNDEFINED_LEGAL
mstatus.TVM	20	RO RW	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
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 RW RO	UNDEFINED_LEGAL
mstatus.MPP	12:11	RW-H	UNDEFINED_LEGAL
mstatus.VS	10:9	RW-H RO RW RO	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.25.4. Fields

mstatus.SD Field

Location:

63

Description:

State Dirty.

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.MDT Field

MDT is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

42

Description:

Machine Disable Trap

Written to 1 when entering M-mode from an exception/interrupt.
When returning via an MRET instruction, the bit is written to 0.
On reset in set to 1, and software should write it to 0 when boot sequence is done.
When mstatus.MDT=1, direct write by CSR instruction cannot set mstatus.MIE to 1, if not written together.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

mstatus.MPV Field

MPV is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

39

Description:

Machine Previous Virtualization mode

Written with the prior virtualization mode when entering M-mode from an exception/interrupt.
When returning via an MRET instruction, the virtualization mode becomes the value of MPV unless MPP=3, in which case the virtualization mode is always 0.
Can also be written by software.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

mstatus.GVA Field

GVA is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

38

Description:

Guest Virtual Address

When a trap is taken and a guest virtual address is written into mtval, GVA is set.
When a trap is taken and a guest virtual address is written into mtval, GVA is cleared.

Type:

RW-H

Reset value:

`0`

mstatus.MBE Field

MBE is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

37

Description:

M-mode Big Endian

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

[when,"M_MODE_ENDIANNESS == little"]
Since the CPU does not support big endian, this is hardwired to 0.

[when,"M_MODE_ENDIANNESS == big"]
Since the CPU does not support little endian, this is hardwired to 1.

Type:

RO

Reset value:

0

mstatus.SBE Field

SBE is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

36

Description:

S-mode Big Endian

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

[when,"S_MODE_ENDIANNESS == little"]
Since the CPU does not support big endian, this is hardwired to 0.

[when,"S_MODE_ENDIANNESS == big"]
Since the CPU does not support little endian, this is hardwired to 1.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.SXL Field

SXL is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

35:34

Description:

S-mode XLEN

Sets the effective XLEN for S-mode (0 = 32-bit, 1 = 64-bit, 2 = 128-bit [reserved]).

[when,"SXLEN==32"]
Since the CPU only supports SXLEN==32, this is hardwired to 0.

[when,"SXLEN==32"]
Since the CPU only supports SXLEN==64, this is hardwired to 1.

[when,"SXLEN=3264"] + — + It is not valid to have SXLEN less than UXLEN.

It is UNDEFINED_LEGAL what will happen if a software sets mstatus.SXL to be greater than mstatus.UXL.

It is UNDEFINED_LEGAL to set the MSB of SXL. + — +

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.UXL Field

UXL is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

33:32

Description:

U-mode XLEN.

Sets the effective XLEN for U-mode (`0` = 32-bit, 1 = 64-bit, 2 = 128-bit [reserved]).

[when,"UXLEN == 32"]
Since the CPU only supports UXLEN==32, this is hardwired to `0`.

[when,"UXLEN == 64"]
Since the CPU only supports UXLEN==64, this is hardwired to 1.

[when,"UXLEN == 3264"] + — + It is not valid to have SXLEN less than UXLEN.

It is UNDEFINED_LEGAL what will happen if a software sets mstatus.SXL to be greater than mstatus.UXL.

It is UNDEFINED_LEGAL to set the MSB of UXL. + — +

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.TSR Field

Location:

22

Description:

Trap SRET

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:

Timeout Wait

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:

Trap Virtual Memory.

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:

Make eXecutable Readable.

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:

permit Supervisor Memory Access.

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:

Modify PRiVilege.

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:

Custom (X) extension context Status

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:

Floating point context status.

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

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.MPP Field

Location:

12:11

Description:

M-mode Previous Privilege.

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:

UNDEFINED_LEGAL

mstatus.VS Field

Location:

10:9

Description:

Vector context status.

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

RW

RO

Reset value:

UNDEFINED_LEGAL

mstatus.SPP Field

Location:

8

Description:

S-mode Previous Privilege

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:

M-mode Previous Interrupt Enable

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:

U-mode Big Endian

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:

S-mode Previous Interrupt Enable

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:

M-mode Interrupt Enable

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:

S-mode Interrupt Enable

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.25.5. 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
MDT = csr_value.MDT
MPV = csr_value.MPV
GVA = csr_value.GVA
MBE = csr_value.MBE
SBE = csr_value.SBE
SXL = if (csr_value.SXL < csr_value.UXL) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (csr_value.SXL > 1) {
  # SXL != [0, 1] is not defined (2 reserved for RV128, but that isn't ratified)
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else {
  return csr_value.SXL;
}

UXL = if (csr_value.SXL < csr_value.UXL) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (csr_value.UXL > 1) {
  # UXL != [0, 1] is not defined (2 reserved for RV128, but that isn't ratified)
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else {
  return csr_value.UXL;
}

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 (CSR[misa].F == 1'b1){
  return ary_includes?<$array_size(MSTATUS_FS_LEGAL_VALUES), 2>(MSTATUS_FS_LEGAL_VALUES, csr_value.FS) ? csr_value.FS :
UNDEFINED_LEGAL_DETERMINISTIC;
} else if ((CSR[misa].S == 1'b0) && (CSR[misa].F == 1'b0)) {
  # must be read-only-0
  return 0;
} else {
  # there will be no hardware update in this case because we know the F extension isn't implemented
  return ary_includes?<$array_size(MSTATUS_FS_LEGAL_VALUES), 2>(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 (CSR[misa].V == 1'b1){
  return ary_includes?<$array_size(MSTATUS_VS_LEGAL_VALUES), 2>(MSTATUS_VS_LEGAL_VALUES, csr_value.FS) ? csr_value.FS :
UNDEFINED_LEGAL_DETERMINISTIC;
} else if ((CSR[misa].S == 1'b0) && (CSR[misa].V == 1'b0)) {
  # must be read-only-0
  return 0;
} else {
  # there will be no hardware update in this case because we know the V extension isn't implemented
  return ary_includes?<$array_size(MSTATUS_VS_LEGAL_VALUES), 2>(MSTATUS_VS_LEGAL_VALUES, csr_value.FS) ? csr_value.FS :
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.26. mtval

Machine Trap Value

Holds trap-specific information

C.26.1. Attributes

CSR Address	0x343
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.26.2. Format

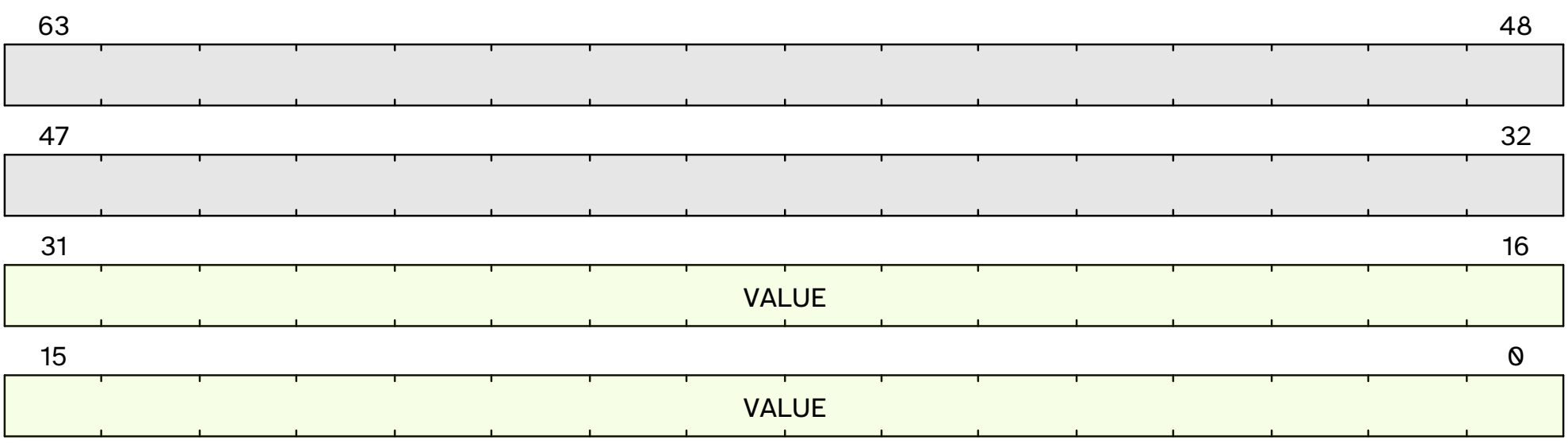


Figure 26. mtval format

C.26.3. Field Summary

Name	Location	Type	Reset Value
mtval.VALUE	63:0	RW-H	0

C.26.4. Fields

mtval.VALUE Field

Location:

63:0

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.27. mtvec

Machine Trap Vector Control

Controls where traps jump.

C.27.1. Attributes

CSR Address	0x305
Defining extension	• Sm, version >= Sm@1.11.0
Length	64-bit
Privilege Mode	M

C.27.2. Format

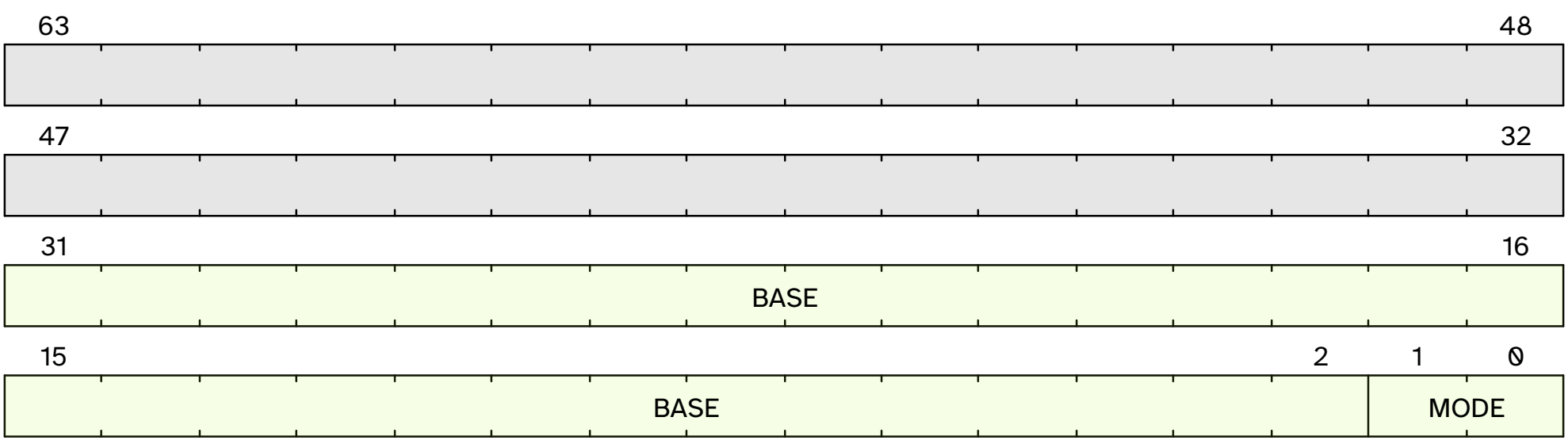


Figure 27. mtvec format

C.27.3. Field Summary

Name	Location	Type	Reset Value
mtvec.BASE	63:2	RW-R	0
mtvec.MODE	1:0	RW-R	UNDEFINED_LEGAL

C.27.4. Fields

mtvec.BASE Field

Location:

63:2

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:

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:

RW-R

Reset value:

UNDEFINED_LEGAL

C.27.5. 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
return csr_value.BASE;

MODE = if (csr_value.MODE == 0) {
  if (ary_includes?<$array_size(MTVEC_MODES), 2>(MTVEC_MODES, 0)) {
    return csr_value.MODE;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else if (csr_value.MODE == 1) {
  if (ary_includes?<$array_size(MTVEC_MODES), 2>(MTVEC_MODES, 1)) {
    return csr_value.MODE;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}
```

C.28. mvendorid

Machine Vendor ID

Reports the JEDEC manufacturer ID of the core.

C.28.1. Attributes

CSR Address	0xf11
Defining extension	• Sm, version >= Sm@1.11.0
Length	32-bit
Privilege Mode	M

C.28.2. Format

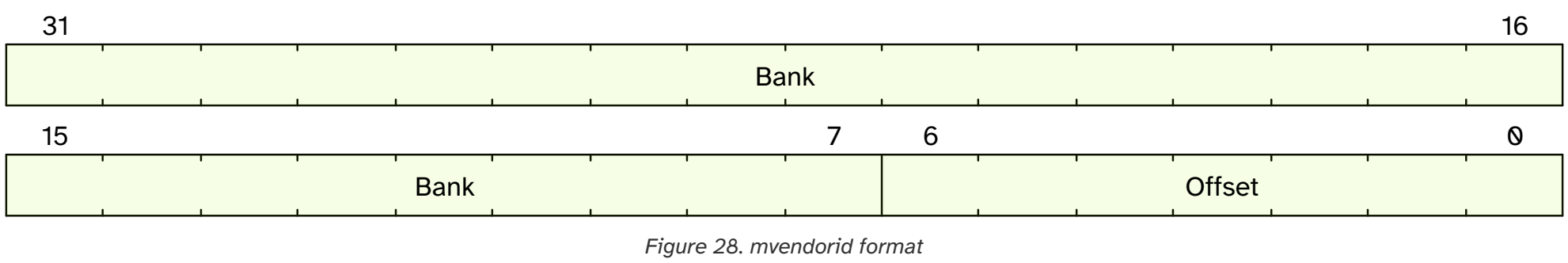


Figure 28. mvendorid format

C.28.3. Field Summary

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

C.28.4. 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.29. pmpaddr0

PMP Address 0

PMP entry address

C.29.1. Attributes

CSR Address	0x3b0
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.29.2. Format

Figure 29. pmpaddr0 format

C.29.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr0.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.29.4. Fields

pmpaddr0.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 0 (or, if `pmp1cfg.A == TOR`, for PMP entry 1).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.29.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 0) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.29.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp0cfg[4] == 1)) {
    return pmpaddr0.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp0cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr0.ADDR & ~mask;
  } else {
    return pmpaddr0.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp0cfg[4] == 1)) {
    return pmpaddr0.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp0cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr0.ADDR & ~mask;
  } else {
    return pmpaddr0.ADDR;
  }
}
```

C.30. pmpaddr1

PMP Address 1

PMP entry address

C.30.1. Attributes

CSR Address	0x3b1
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.30.2. Format

Figure 30. pmpaddr1 format

C.30.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr1.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.30.4. Fields

pmpaddr1.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 1 (or, if `pmp2cfg.A == TOR`, for PMP entry 2).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.30.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 1) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.30.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp1cfg[4] == 1)) {
    return pmpaddr1.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp1cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr1.ADDR & ~mask;
  } else {
    return pmpaddr1.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp1cfg[4] == 1)) {
    return pmpaddr1.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp1cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr1.ADDR & ~mask;
  } else {
    return pmpaddr1.ADDR;
  }
}
```

C.31. pmpaddr10

PMP Address 10

PMP entry address

C.31.1. Attributes

CSR Address	0x3ba
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.31.2. Format

Figure 31. pmpaddr10 format

C.31.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr10.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.31.4. Fields

pmpaddr10.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 10 (or, if `pmp11cfg.A` == TOR, for PMP entry 11).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.31.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 10) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.31.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp10cfg[4] == 1)) {
    return pmpaddr10.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp10cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr10.ADDR & ~mask;
  } else {
    return pmpaddr10.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp10cfg[4] == 1)) {
    return pmpaddr10.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp10cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr10.ADDR & ~mask;
  } else {
    return pmpaddr10.ADDR;
  }
}
```

C.32. pmpaddr11

PMP Address 11

PMP entry address

C.32.1. Attributes

CSR Address	0x3bb
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.32.2. Format

Figure 32. pmpaddr11 format

C.32.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr11.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.32.4. Fields

pmpaddr11.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 11 (or, if `pmp12cfg.A` == TOR, for PMP entry 12).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.32.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 11) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.32.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp11cfg[4] == 1)) {
    return pmpaddr11.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp11cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr11.ADDR & ~mask;
  } else {
    return pmpaddr11.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp11cfg[4] == 1)) {
    return pmpaddr11.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp11cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr11.ADDR & ~mask;
  } else {
    return pmpaddr11.ADDR;
  }
}
```

C.33. pmpaddr12

PMP Address 12

PMP entry address

C.33.1. Attributes

CSR Address	0x3bc
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.33.2. Format

Figure 33. pmpaddr12 format

C.33.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr12.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.33.4. Fields

pmpaddr12.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 12 (or, if `pmp13cfg.A` == TOR, for PMP entry 13).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.33.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 12) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.33.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg3.pmp12cfg[4] == 1)) {
    return pmpaddr12.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg3.pmp12cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr12.ADDR & ~mask;
  } else {
    return pmpaddr12.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp12cfg[4] == 1)) {
    return pmpaddr12.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp12cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr12.ADDR & ~mask;
  } else {
    return pmpaddr12.ADDR;
  }
}
```

C.34. pmpaddr13

PMP Address 13

PMP entry address

C.34.1. Attributes

CSR Address	0x3bd
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.34.2. Format

Figure 34. pmpaddr13 format

C.34.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr13.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.34.4. Fields

pmpaddr13.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 13 (or, if `pmp14cfg.A` == TOR, for PMP entry 14).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.34.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 13) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.34.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg3.pmp13cfg[4] == 1)) {
    return pmpaddr13.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg3.pmp13cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr13.ADDR & ~mask;
  } else {
    return pmpaddr13.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp13cfg[4] == 1)) {
    return pmpaddr13.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp13cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr13.ADDR & ~mask;
  } else {
    return pmpaddr13.ADDR;
  }
}
```

C.35. pmpaddr14

PMP Address 14

PMP entry address

C.35.1. Attributes

CSR Address	0x3be
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.35.2. Format

Figure 35. pmpaddr14 format

C.35.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr14.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.35.4. Fields

pmpaddr14.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 14 (or, if `pmp15cfg.A` == TOR, for PMP entry 15).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.35.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 14) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.35.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg3.pmp14cfg[4] == 1)) {
    return pmpaddr14.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg3.pmp14cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr14.ADDR & ~mask;
  } else {
    return pmpaddr14.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp14cfg[4] == 1)) {
    return pmpaddr14.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp14cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr14.ADDR & ~mask;
  } else {
    return pmpaddr14.ADDR;
  }
}
```

C.36. pmpaddr15

PMP Address 15

PMP entry address

C.36.1. Attributes

CSR Address	0x3bf
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.36.2. Format

Figure 36. pmpaddr15 format

C.36.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr15.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.36.4. Fields

pmpaddr15.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 15 (or, if `pmp16cfg.A` == TOR, for PMP entry 16).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.36.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 15) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.36.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg3.pmp15cfg[4] == 1)) {
    return pmpaddr15.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg3.pmp15cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr15.ADDR & ~mask;
  } else {
    return pmpaddr15.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp15cfg[4] == 1)) {
    return pmpaddr15.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp15cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr15.ADDR & ~mask;
  } else {
    return pmpaddr15.ADDR;
  }
}
```

C.37. pmpaddr16

PMP Address 16

PMP entry address

C.37.1. Attributes

CSR Address	0x3c0
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.37.2. Format

Figure 37. pmpaddr16 format

C.37.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr16.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.37.4. Fields

pmpaddr16.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 16 (or, if `pmp17cfg.A` == TOR, for PMP entry 17).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.37.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 16) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.37.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp16cfg[4] == 1)) {
    return pmpaddr16.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp16cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr16.ADDR & ~mask;
  } else {
    return pmpaddr16.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp16cfg[4] == 1)) {
    return pmpaddr16.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp16cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr16.ADDR & ~mask;
  } else {
    return pmpaddr16.ADDR;
  }
}
```

C.38. pmpaddr17

PMP Address 17

PMP entry address

C.38.1. Attributes

CSR Address	0x3c1
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.38.2. Format

Figure 38. pmpaddr17 format

C.38.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr17.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.38.4. Fields

pmpaddr17.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 17 (or, if `pmp18cfg.A` == TOR, for PMP entry 18).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.38.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 17) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.38.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp17cfg[4] == 1)) {
    return pmpaddr17.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp17cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr17.ADDR & ~mask;
  } else {
    return pmpaddr17.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp17cfg[4] == 1)) {
    return pmpaddr17.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp17cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr17.ADDR & ~mask;
  } else {
    return pmpaddr17.ADDR;
  }
}
```

C.39. pmpaddr18

PMP Address 18

PMP entry address

C.39.1. Attributes

CSR Address	0x3c2
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.39.2. Format

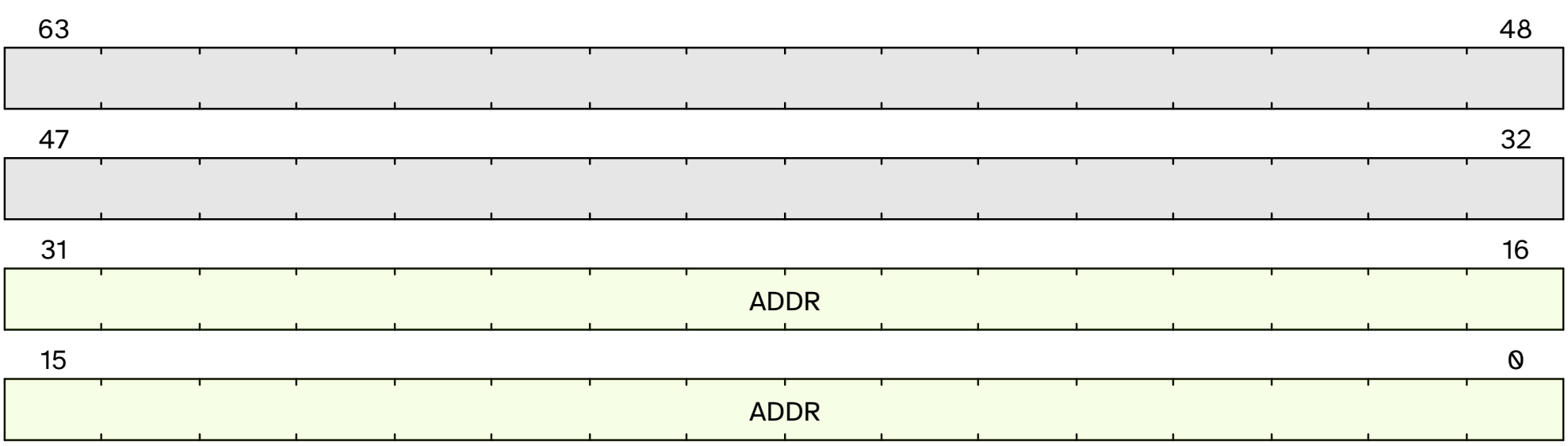


Figure 39. pmpaddr18 format

C.39.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr18.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.39.4. Fields

pmpaddr18.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 18 (or, if `pmp19cfg.A` == TOR, for PMP entry 19).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.39.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 18) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.39.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp18cfg[4] == 1)) {
    return pmpaddr18.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp18cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr18.ADDR & ~mask;
  } else {
    return pmpaddr18.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp18cfg[4] == 1)) {
    return pmpaddr18.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp18cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr18.ADDR & ~mask;
  } else {
    return pmpaddr18.ADDR;
  }
}
```

C.40. pmpaddr19

PMP Address 19

PMP entry address

C.40.1. Attributes

CSR Address	0x3c3
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.40.2. Format

Figure 40. pmpaddr19 format

C.40.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr19.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.40.4. Fields

pmpaddr19.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 19 (or, if `pmp20cfg.A` == TOR, for PMP entry 20).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.40.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 19) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.40.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp19cfg[4] == 1)) {
    return pmpaddr19.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp19cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr19.ADDR & ~mask;
  } else {
    return pmpaddr19.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp19cfg[4] == 1)) {
    return pmpaddr19.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp19cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr19.ADDR & ~mask;
  } else {
    return pmpaddr19.ADDR;
  }
}
```

C.41. pmpaddr2

PMP Address 2

PMP entry address

C.41.1. Attributes

CSR Address	0x3b2
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.41.2. Format

Figure 41. pmpaddr2 format

C.41.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr2.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.41.4. Fields

pmpaddr2.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 2 (or, if `pmp3cfg.A == TOR`, for PMP entry 3).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.41.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 2) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.41.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp2cfg[4] == 1)) {
    return pmpaddr2.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp2cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr2.ADDR & ~mask;
  } else {
    return pmpaddr2.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp2cfg[4] == 1)) {
    return pmpaddr2.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp2cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr2.ADDR & ~mask;
  } else {
    return pmpaddr2.ADDR;
  }
}
```

C.42. pmpaddr20

PMP Address 20

PMP entry address

C.42.1. Attributes

CSR Address	0x3c4
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.42.2. Format

Figure 42. pmpaddr20 format

C.42.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr20.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.42.4. Fields

pmpaddr20.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 20 (or, if `pmp21cfg.A` == TOR, for PMP entry 21).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.42.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 20) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.42.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg5.pmp20cfg[4] == 1)) {
    return pmpaddr20.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg5.pmp20cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr20.ADDR & ~mask;
  } else {
    return pmpaddr20.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp20cfg[4] == 1)) {
    return pmpaddr20.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp20cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr20.ADDR & ~mask;
  } else {
    return pmpaddr20.ADDR;
  }
}
```

C.43. pmpaddr21

PMP Address 21

PMP entry address

C.43.1. Attributes

CSR Address	0x3c5
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.43.2. Format

Figure 43. pmpaddr21 format

C.43.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr21.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.43.4. Fields

pmpaddr21.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 21 (or, if `pmp22cfg.A` == TOR, for PMP entry 22).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.43.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 21) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.43.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg5.pmp21cfg[4] == 1)) {
    return pmpaddr21.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg5.pmp21cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr21.ADDR & ~mask;
  } else {
    return pmpaddr21.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp21cfg[4] == 1)) {
    return pmpaddr21.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp21cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr21.ADDR & ~mask;
  } else {
    return pmpaddr21.ADDR;
  }
}
```

C.44. pmpaddr22

PMP Address 22

PMP entry address

C.44.1. Attributes

CSR Address	0x3c6
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.44.2. Format

Figure 44. pmpaddr22 format

C.44.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr22.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.44.4. Fields

pmpaddr22.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 22 (or, if `pmp23cfg.A` == TOR, for PMP entry 23).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.44.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 22) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.44.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg5.pmp22cfg[4] == 1)) {
    return pmpaddr22.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg5.pmp22cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr22.ADDR & ~mask;
  } else {
    return pmpaddr22.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp22cfg[4] == 1)) {
    return pmpaddr22.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp22cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr22.ADDR & ~mask;
  } else {
    return pmpaddr22.ADDR;
  }
}
```

C.45. pmpaddr23

PMP Address 23

PMP entry address

C.45.1. Attributes

CSR Address	0x3c7
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.45.2. Format

Figure 45. pmpaddr23 format

C.45.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr23.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.45.4. Fields

pmpaddr23.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 23 (or, if `pmp24cfg.A` == TOR, for PMP entry 24).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.45.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 23) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.45.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg5.pmp23cfg[4] == 1)) {
    return pmpaddr23.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg5.pmp23cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr23.ADDR & ~mask;
  } else {
    return pmpaddr23.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg4.pmp23cfg[4] == 1)) {
    return pmpaddr23.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg4.pmp23cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr23.ADDR & ~mask;
  } else {
    return pmpaddr23.ADDR;
  }
}
```

C.46. pmpaddr24

PMP Address 24

PMP entry address

C.46.1. Attributes

CSR Address	0x3c8
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.46.2. Format

Figure 46. pmpaddr24 format

C.46.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr24.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.46.4. Fields

pmpaddr24.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 24 (or, if `pmp25cfg.A` == TOR, for PMP entry 25).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.46.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 24) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.46.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp24cfg[4] == 1)) {
    return pmpaddr24.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp24cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr24.ADDR & ~mask;
  } else {
    return pmpaddr24.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp24cfg[4] == 1)) {
    return pmpaddr24.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp24cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr24.ADDR & ~mask;
  } else {
    return pmpaddr24.ADDR;
  }
}
```

C.47. pmpaddr25

PMP Address 25

PMP entry address

C.47.1. Attributes

CSR Address	0x3c9
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.47.2. Format

Figure 47. pmpaddr25 format

C.47.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr25.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.47.4. Fields

pmpaddr25.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 25 (or, if `pmp26cfg.A` == TOR, for PMP entry 26).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.47.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 25) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.47.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp25cfg[4] == 1)) {
    return pmpaddr25.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp25cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr25.ADDR & ~mask;
  } else {
    return pmpaddr25.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp25cfg[4] == 1)) {
    return pmpaddr25.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp25cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr25.ADDR & ~mask;
  } else {
    return pmpaddr25.ADDR;
  }
}
```

C.48. pmpaddr26

PMP Address 26

PMP entry address

C.48.1. Attributes

CSR Address	0x3ca
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.48.2. Format

Figure 48. pmpaddr26 format

C.48.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr26.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.48.4. Fields

pmpaddr26.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 26 (or, if `pmp27cfg.A` == TOR, for PMP entry 27).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.48.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 26) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.48.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp26cfg[4] == 1)) {
    return pmpaddr26.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp26cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr26.ADDR & ~mask;
  } else {
    return pmpaddr26.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp26cfg[4] == 1)) {
    return pmpaddr26.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp26cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr26.ADDR & ~mask;
  } else {
    return pmpaddr26.ADDR;
  }
}
```

C.49. pmpaddr27

PMP Address 27

PMP entry address

C.49.1. Attributes

CSR Address	0x3cb
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.49.2. Format

Figure 49. pmpaddr27 format

C.49.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr27.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.49.4. Fields

pmpaddr27.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 27 (or, if `pmp28cfg.A` == TOR, for PMP entry 28).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.49.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 27) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.49.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp27cfg[4] == 1)) {
    return pmpaddr27.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp27cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr27.ADDR & ~mask;
  } else {
    return pmpaddr27.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp27cfg[4] == 1)) {
    return pmpaddr27.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp27cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr27.ADDR & ~mask;
  } else {
    return pmpaddr27.ADDR;
  }
}
```

C.50. pmpaddr28

PMP Address 28

PMP entry address

C.50.1. Attributes

CSR Address	0x3cc
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.50.2. Format

Figure 50. pmpaddr28 format

C.50.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr28.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.50.4. Fields

pmpaddr28.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 28 (or, if `pmp29cfg.A` == TOR, for PMP entry 29).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.50.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 28) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.50.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg7.pmp28cfg[4] == 1)) {
    return pmpaddr28.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg7.pmp28cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr28.ADDR & ~mask;
  } else {
    return pmpaddr28.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp28cfg[4] == 1)) {
    return pmpaddr28.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp28cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr28.ADDR & ~mask;
  } else {
    return pmpaddr28.ADDR;
  }
}
```

C.51. pmpaddr29

PMP Address 29

PMP entry address

C.51.1. Attributes

CSR Address	0x3cd
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.51.2. Format

Figure 51. pmpaddr29 format

C.51.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr29.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.51.4. Fields

pmpaddr29.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 29 (or, if `pmp30cfg.A` == TOR, for PMP entry 30).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.51.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 29) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.51.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg7.pmp29cfg[4] == 1)) {
    return pmpaddr29.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg7.pmp29cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr29.ADDR & ~mask;
  } else {
    return pmpaddr29.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp29cfg[4] == 1)) {
    return pmpaddr29.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp29cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr29.ADDR & ~mask;
  } else {
    return pmpaddr29.ADDR;
  }
}
```

C.52. pmpaddr3

PMP Address 3

PMP entry address

C.52.1. Attributes

CSR Address	0x3b3
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.52.2. Format

Figure 52. pmpaddr3 format

C.52.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr3.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.52.4. Fields

pmpaddr3.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 3 (or, if `pmp4cfg.A == TOR`, for PMP entry 4).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.52.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 3) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.52.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp3cfg[4] == 1)) {
    return pmpaddr3.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp3cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr3.ADDR & ~mask;
  } else {
    return pmpaddr3.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp3cfg[4] == 1)) {
    return pmpaddr3.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp3cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr3.ADDR & ~mask;
  } else {
    return pmpaddr3.ADDR;
  }
}
```

C.53. pmpaddr30

PMP Address 30

PMP entry address

C.53.1. Attributes

CSR Address	0x3ce
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.53.2. Format

Figure 53. pmpaddr30 format

C.53.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr30.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.53.4. Fields

pmpaddr30.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 30 (or, if `pmp31cfg.A` == TOR, for PMP entry 31).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.53.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 30) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.53.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg7.pmp30cfg[4] == 1)) {
    return pmpaddr30.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg7.pmp30cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr30.ADDR & ~mask;
  } else {
    return pmpaddr30.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp30cfg[4] == 1)) {
    return pmpaddr30.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp30cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr30.ADDR & ~mask;
  } else {
    return pmpaddr30.ADDR;
  }
}
```

C.54. pmpaddr31

PMP Address 31

PMP entry address

C.54.1. Attributes

CSR Address	0x3cf
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.54.2. Format

Figure 54. pmpaddr31 format

C.54.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr31.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.54.4. Fields

pmpaddr31.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 31 (or, if `pmp32cfg.A` == TOR, for PMP entry 32).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.54.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 31) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.54.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg7.pmp31cfg[4] == 1)) {
    return pmpaddr31.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg7.pmp31cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr31.ADDR & ~mask;
  } else {
    return pmpaddr31.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg6.pmp31cfg[4] == 1)) {
    return pmpaddr31.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg6.pmp31cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr31.ADDR & ~mask;
  } else {
    return pmpaddr31.ADDR;
  }
}
```

C.55. pmpaddr32

PMP Address 32

PMP entry address

C.55.1. Attributes

CSR Address	0x3d0
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.55.2. Format

Figure 55. pmpaddr32 format

C.55.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr32.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.55.4. Fields

pmpaddr32.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 32 (or, if `pmp33cfg.A` == TOR, for PMP entry 33).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.55.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 32) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.55.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp32cfg[4] == 1)) {
    return pmpaddr32.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp32cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr32.ADDR & ~mask;
  } else {
    return pmpaddr32.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp32cfg[4] == 1)) {
    return pmpaddr32.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp32cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr32.ADDR & ~mask;
  } else {
    return pmpaddr32.ADDR;
  }
}
```

C.56. pmpaddr33

PMP Address 33

PMP entry address

C.56.1. Attributes

CSR Address	0x3d1
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.56.2. Format

Figure 56. pmpaddr33 format

C.56.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr33.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.56.4. Fields

pmpaddr33.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 33 (or, if `pmp34cfg.A == TOR`, for PMP entry 34).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.56.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 33) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.56.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp33cfg[4] == 1)) {
    return pmpaddr33.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp33cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr33.ADDR & ~mask;
  } else {
    return pmpaddr33.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp33cfg[4] == 1)) {
    return pmpaddr33.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp33cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr33.ADDR & ~mask;
  } else {
    return pmpaddr33.ADDR;
  }
}
```

C.57. pmpaddr34

PMP Address 34

PMP entry address

C.57.1. Attributes

CSR Address	0x3d2
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.57.2. Format

Figure 57. pmpaddr34 format

C.57.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr34.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.57.4. Fields

pmpaddr34.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 34 (or, if `pmp35cfg.A` == TOR, for PMP entry 35).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.57.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 34) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.57.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp34cfg[4] == 1)) {
    return pmpaddr34.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp34cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr34.ADDR & ~mask;
  } else {
    return pmpaddr34.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp34cfg[4] == 1)) {
    return pmpaddr34.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp34cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr34.ADDR & ~mask;
  } else {
    return pmpaddr34.ADDR;
  }
}
```

C.58. pmpaddr35

PMP Address 35

PMP entry address

C.58.1. Attributes

CSR Address	0x3d3
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.58.2. Format

Figure 58. pmpaddr35 format

C.58.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr35.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.58.4. Fields

pmpaddr35.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 35 (or, if `pmp36cfg.A` == TOR, for PMP entry 36).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.58.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 35) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.58.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp35cfg[4] == 1)) {
    return pmpaddr35.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp35cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr35.ADDR & ~mask;
  } else {
    return pmpaddr35.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp35cfg[4] == 1)) {
    return pmpaddr35.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp35cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr35.ADDR & ~mask;
  } else {
    return pmpaddr35.ADDR;
  }
}
```

C.59. pmpaddr36

PMP Address 36

PMP entry address

C.59.1. Attributes

CSR Address	0x3d4
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.59.2. Format

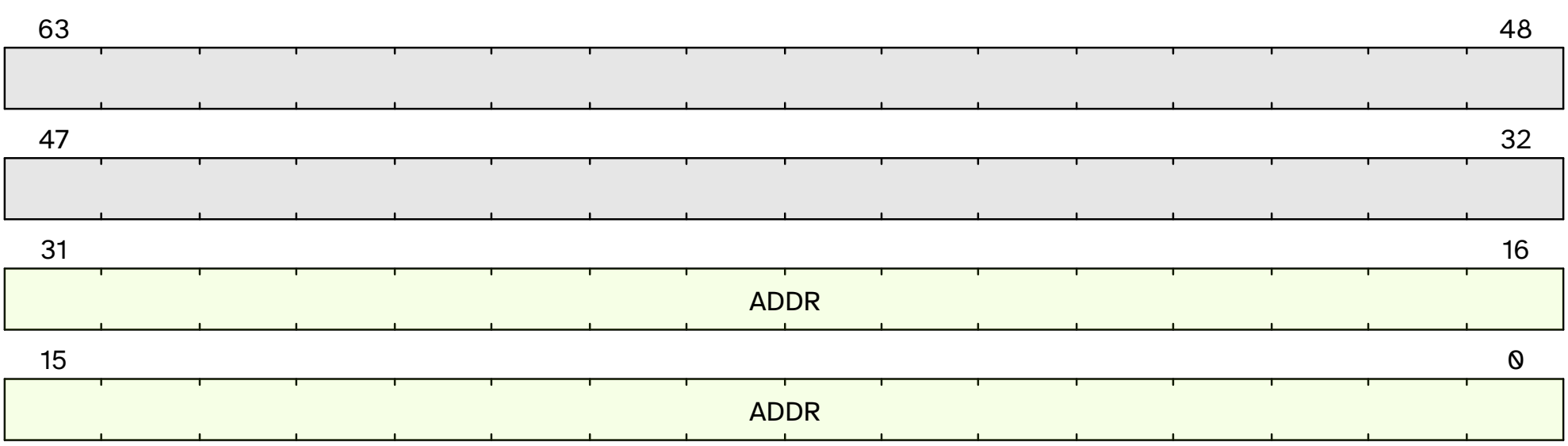


Figure 59. pmpaddr36 format

C.59.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr36.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.59.4. Fields

pmpaddr36.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 36 (or, if `pmp37cfg.A` == TOR, for PMP entry 37).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.59.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 36) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.59.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg9.pmp36cfg[4] == 1)) {
    return pmpaddr36.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg9.pmp36cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr36.ADDR & ~mask;
  } else {
    return pmpaddr36.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp36cfg[4] == 1)) {
    return pmpaddr36.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp36cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr36.ADDR & ~mask;
  } else {
    return pmpaddr36.ADDR;
  }
}
```

C.60. pmpaddr37

PMP Address 37

PMP entry address

C.60.1. Attributes

CSR Address	0x3d5
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.60.2. Format

Figure 60. pmpaddr37 format

C.60.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr37.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.60.4. Fields

pmpaddr37.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 37 (or, if `pmp38cfg.A` == TOR, for PMP entry 38).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.60.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 37) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.60.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg9.pmp37cfg[4] == 1)) {
    return pmpaddr37.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg9.pmp37cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr37.ADDR & ~mask;
  } else {
    return pmpaddr37.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp37cfg[4] == 1)) {
    return pmpaddr37.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp37cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr37.ADDR & ~mask;
  } else {
    return pmpaddr37.ADDR;
  }
}
```

C.61. pmpaddr38

PMP Address 38

PMP entry address

C.61.1. Attributes

CSR Address	0x3d6
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.61.2. Format

Figure 61. pmpaddr38 format

C.61.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr38.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.61.4. Fields

pmpaddr38.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 38 (or, if `pmp39cfg.A` == TOR, for PMP entry 39).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.61.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 38) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.61.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg9.pmp38cfg[4] == 1)) {
    return pmpaddr38.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg9.pmp38cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr38.ADDR & ~mask;
  } else {
    return pmpaddr38.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp38cfg[4] == 1)) {
    return pmpaddr38.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp38cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr38.ADDR & ~mask;
  } else {
    return pmpaddr38.ADDR;
  }
}
```

C.62. pmpaddr39

PMP Address 39

PMP entry address

C.62.1. Attributes

CSR Address	0x3d7
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.62.2. Format

Figure 62. pmpaddr39 format

C.62.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr39.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.62.4. Fields

pmpaddr39.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 39 (or, if `pmp40cfg.A` == TOR, for PMP entry 40).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.62.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 39) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.62.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg9.pmp39cfg[4] == 1)) {
    return pmpaddr39.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg9.pmp39cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr39.ADDR & ~mask;
  } else {
    return pmpaddr39.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg8.pmp39cfg[4] == 1)) {
    return pmpaddr39.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg8.pmp39cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr39.ADDR & ~mask;
  } else {
    return pmpaddr39.ADDR;
  }
}
```

C.63. pmpaddr4

PMP Address 4

PMP entry address

C.63.1. Attributes

CSR Address	0x3b4
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.63.2. Format

Figure 63. pmpaddr4 format

C.63.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr4.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.63.4. Fields

pmpaddr4.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 4 (or, if `pmp5cfg.A == TOR`, for PMP entry 5).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.63.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 4) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.63.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg1.pmp4cfg[4] == 1)) {
    return pmpaddr4.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg1.pmp4cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr4.ADDR & ~mask;
  } else {
    return pmpaddr4.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp4cfg[4] == 1)) {
    return pmpaddr4.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp4cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr4.ADDR & ~mask;
  } else {
    return pmpaddr4.ADDR;
  }
}
```

C.64. pmpaddr40

PMP Address 40

PMP entry address

C.64.1. Attributes

CSR Address	0x3d8
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.64.2. Format

Figure 64. pmpaddr40 format

C.64.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr40.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.64.4. Fields

pmpaddr40.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 40 (or, if `pmp41cfg.A` == TOR, for PMP entry 41).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.64.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 40) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.64.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp40cfg[4] == 1)) {
    return pmpaddr40.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp40cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr40.ADDR & ~mask;
  } else {
    return pmpaddr40.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp40cfg[4] == 1)) {
    return pmpaddr40.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp40cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr40.ADDR & ~mask;
  } else {
    return pmpaddr40.ADDR;
  }
}
```

C.65. pmpaddr41

PMP Address 41

PMP entry address

C.65.1. Attributes

CSR Address	0x3d9
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.65.2. Format

Figure 65. pmpaddr41 format

C.65.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr41.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.65.4. Fields

pmpaddr41.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 41 (or, if `pmp42cfg.A` == TOR, for PMP entry 42).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.65.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 41) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.65.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp41cfg[4] == 1)) {
    return pmpaddr41.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp41cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr41.ADDR & ~mask;
  } else {
    return pmpaddr41.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp41cfg[4] == 1)) {
    return pmpaddr41.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp41cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr41.ADDR & ~mask;
  } else {
    return pmpaddr41.ADDR;
  }
}
```

C.66. pmpaddr42

PMP Address 42

PMP entry address

C.66.1. Attributes

CSR Address	0x3da
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.66.2. Format

Figure 66. pmpaddr42 format

C.66.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr42.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.66.4. Fields

pmpaddr42.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 42 (or, if `pmp43cfg.A` == TOR, for PMP entry 43).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.66.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 42) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.66.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp42cfg[4] == 1)) {
    return pmpaddr42.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp42cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr42.ADDR & ~mask;
  } else {
    return pmpaddr42.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp42cfg[4] == 1)) {
    return pmpaddr42.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp42cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr42.ADDR & ~mask;
  } else {
    return pmpaddr42.ADDR;
  }
}
```

C.67. pmpaddr43

PMP Address 43

PMP entry address

C.67.1. Attributes

CSR Address	0x3db
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.67.2. Format

Figure 67. pmpaddr43 format

C.67.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr43.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.67.4. Fields

pmpaddr43.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 43 (or, if `pmp44cfg.A` == TOR, for PMP entry 44).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.67.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 43) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.67.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp43cfg[4] == 1)) {
    return pmpaddr43.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp43cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr43.ADDR & ~mask;
  } else {
    return pmpaddr43.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp43cfg[4] == 1)) {
    return pmpaddr43.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp43cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr43.ADDR & ~mask;
  } else {
    return pmpaddr43.ADDR;
  }
}
```

C.68. pmpaddr44

PMP Address 44

PMP entry address

C.68.1. Attributes

CSR Address	0x3dc
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.68.2. Format

Figure 68. pmpaddr44 format

C.68.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr44.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.68.4. Fields

pmpaddr44.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 44 (or, if `pmp45cfg.A` == TOR, for PMP entry 45).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.68.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 44) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.68.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg11.pmp44cfg[4] == 1)) {
    return pmpaddr44.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg11.pmp44cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr44.ADDR & ~mask;
  } else {
    return pmpaddr44.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp44cfg[4] == 1)) {
    return pmpaddr44.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp44cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr44.ADDR & ~mask;
  } else {
    return pmpaddr44.ADDR;
  }
}
```

C.69. pmpaddr45

PMP Address 45

PMP entry address

C.69.1. Attributes

CSR Address	0x3dd
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.69.2. Format

Figure 69. pmpaddr45 format

C.69.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr45.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.69.4. Fields

pmpaddr45.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 45 (or, if `pmp46cfg.A` == TOR, for PMP entry 46).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.69.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 45) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.69.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg11.pmp45cfg[4] == 1)) {
    return pmpaddr45.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg11.pmp45cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr45.ADDR & ~mask;
  } else {
    return pmpaddr45.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp45cfg[4] == 1)) {
    return pmpaddr45.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp45cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr45.ADDR & ~mask;
  } else {
    return pmpaddr45.ADDR;
  }
}
```

C.70. pmpaddr46

PMP Address 46

PMP entry address

C.70.1. Attributes

CSR Address	0x3de
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.70.2. Format

Figure 70. pmpaddr46 format

C.70.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr46.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.70.4. Fields

pmpaddr46.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 46 (or, if `pmp47cfg.A` == TOR, for PMP entry 47).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.70.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 46) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.70.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg11.pmp46cfg[4] == 1)) {
    return pmpaddr46.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg11.pmp46cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr46.ADDR & ~mask;
  } else {
    return pmpaddr46.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp46cfg[4] == 1)) {
    return pmpaddr46.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp46cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr46.ADDR & ~mask;
  } else {
    return pmpaddr46.ADDR;
  }
}
```

C.71. pmpaddr47

PMP Address 47

PMP entry address

C.71.1. Attributes

CSR Address	0x3df
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.71.2. Format

Figure 71. pmpaddr47 format

C.71.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr47.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.71.4. Fields

pmpaddr47.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 47 (or, if `pmp48cfg.A` == TOR, for PMP entry 48).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.71.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 47) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.71.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg11.pmp47cfg[4] == 1)) {
    return pmpaddr47.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg11.pmp47cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr47.ADDR & ~mask;
  } else {
    return pmpaddr47.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg10.pmp47cfg[4] == 1)) {
    return pmpaddr47.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg10.pmp47cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr47.ADDR & ~mask;
  } else {
    return pmpaddr47.ADDR;
  }
}
```

C.72. pmpaddr48

PMP Address 48

PMP entry address

C.72.1. Attributes

CSR Address	0x3e0
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.72.2. Format

Figure 72. pmpaddr48 format

C.72.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr48.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.72.4. Fields

pmpaddr48.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 48 (or, if `pmp49cfg.A` == TOR, for PMP entry 49).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.72.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 48) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.72.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp48cfg[4] == 1)) {
    return pmpaddr48.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp48cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr48.ADDR & ~mask;
  } else {
    return pmpaddr48.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp48cfg[4] == 1)) {
    return pmpaddr48.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp48cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr48.ADDR & ~mask;
  } else {
    return pmpaddr48.ADDR;
  }
}
```

C.73. pmpaddr49

PMP Address 49

PMP entry address

C.73.1. Attributes

CSR Address	0x3e1
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.73.2. Format

Figure 73. pmpaddr49 format

C.73.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr49.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.73.4. Fields

pmpaddr49.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 49 (or, if `pmp50cfg.A` == TOR, for PMP entry 50).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.73.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 49) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.73.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp49cfg[4] == 1)) {
    return pmpaddr49.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp49cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr49.ADDR & ~mask;
  } else {
    return pmpaddr49.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp49cfg[4] == 1)) {
    return pmpaddr49.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp49cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr49.ADDR & ~mask;
  } else {
    return pmpaddr49.ADDR;
  }
}
```

C.74. pmpaddr5

PMP Address 5

PMP entry address

C.74.1. Attributes

CSR Address	0x3b5
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.74.2. Format

Figure 74. pmpaddr5 format

C.74.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr5.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.74.4. Fields

pmpaddr5.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 5 (or, if `pmp6cfg.A == TOR`, for PMP entry 6).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.74.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 5) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.74.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg1.pmp5cfg[4] == 1)) {
    return pmpaddr5.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg1.pmp5cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr5.ADDR & ~mask;
  } else {
    return pmpaddr5.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp5cfg[4] == 1)) {
    return pmpaddr5.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp5cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr5.ADDR & ~mask;
  } else {
    return pmpaddr5.ADDR;
  }
}
```

C.75. pmpaddr50

PMP Address 50

PMP entry address

C.75.1. Attributes

CSR Address	0x3e2
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.75.2. Format

Figure 75. pmpaddr50 format

C.75.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr50.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.75.4. Fields

pmpaddr50.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 50 (or, if `pmp51cfg.A` == TOR, for PMP entry 51).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.75.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 50) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.75.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp50cfg[4] == 1)) {
    return pmpaddr50.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp50cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr50.ADDR & ~mask;
  } else {
    return pmpaddr50.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp50cfg[4] == 1)) {
    return pmpaddr50.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp50cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr50.ADDR & ~mask;
  } else {
    return pmpaddr50.ADDR;
  }
}
```

C.76. pmpaddr51

PMP Address 51

PMP entry address

C.76.1. Attributes

CSR Address	0x3e3
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.76.2. Format

Figure 76. pmpaddr51 format

C.76.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr51.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.76.4. Fields

pmpaddr51.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 51 (or, if `pmp52cfg.A` == TOR, for PMP entry 52).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.76.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 51) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.76.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp51cfg[4] == 1)) {
    return pmpaddr51.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp51cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr51.ADDR & ~mask;
  } else {
    return pmpaddr51.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp51cfg[4] == 1)) {
    return pmpaddr51.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp51cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr51.ADDR & ~mask;
  } else {
    return pmpaddr51.ADDR;
  }
}
```

C.77. pmpaddr52

PMP Address 52

PMP entry address

C.77.1. Attributes

CSR Address	0x3e4
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.77.2. Format

Figure 77. pmpaddr52 format

C.77.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr52.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.77.4. Fields

pmpaddr52.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 52 (or, if `pmp53cfg.A` == TOR, for PMP entry 53).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.77.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 52) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.77.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg13.pmp52cfg[4] == 1)) {
    return pmpaddr52.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg13.pmp52cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr52.ADDR & ~mask;
  } else {
    return pmpaddr52.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp52cfg[4] == 1)) {
    return pmpaddr52.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp52cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr52.ADDR & ~mask;
  } else {
    return pmpaddr52.ADDR;
  }
}
```

C.78. pmpaddr53

PMP Address 53

PMP entry address

C.78.1. Attributes

CSR Address	0x3e5
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.78.2. Format

Figure 78. pmpaddr53 format

C.78.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr53.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.78.4. Fields

pmpaddr53.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 53 (or, if `pmp54cfg.A` == TOR, for PMP entry 54).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.78.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 53) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.78.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg13.pmp53cfg[4] == 1)) {
    return pmpaddr53.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg13.pmp53cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr53.ADDR & ~mask;
  } else {
    return pmpaddr53.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp53cfg[4] == 1)) {
    return pmpaddr53.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp53cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr53.ADDR & ~mask;
  } else {
    return pmpaddr53.ADDR;
  }
}
```

C.79. pmpaddr54

PMP Address 54

PMP entry address

C.79.1. Attributes

CSR Address	0x3e6
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.79.2. Format

Figure 79. pmpaddr54 format

C.79.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr54.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.79.4. Fields

pmpaddr54.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 54 (or, if `pmp55cfg.A` == TOR, for PMP entry 55).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.79.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 54) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.79.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg13.pmp54cfg[4] == 1)) {
    return pmpaddr54.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg13.pmp54cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr54.ADDR & ~mask;
  } else {
    return pmpaddr54.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp54cfg[4] == 1)) {
    return pmpaddr54.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp54cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr54.ADDR & ~mask;
  } else {
    return pmpaddr54.ADDR;
  }
}
```

C.80. pmpaddr55

PMP Address 55

PMP entry address

C.80.1. Attributes

CSR Address	0x3e7
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.80.2. Format

Figure 80. pmpaddr55 format

C.80.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr55.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.80.4. Fields

pmpaddr55.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 55 (or, if `pmp56cfg.A` == TOR, for PMP entry 56).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.80.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 55) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.80.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg13.pmp55cfg[4] == 1)) {
    return pmpaddr55.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg13.pmp55cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr55.ADDR & ~mask;
  } else {
    return pmpaddr55.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg12.pmp55cfg[4] == 1)) {
    return pmpaddr55.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg12.pmp55cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr55.ADDR & ~mask;
  } else {
    return pmpaddr55.ADDR;
  }
}
```

C.81. pmpaddr56

PMP Address 56

PMP entry address

C.81.1. Attributes

CSR Address	0x3e8
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.81.2. Format

Figure 81. pmpaddr56 format

C.81.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr56.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.81.4. Fields

pmpaddr56.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 56 (or, if `pmp57cfg.A` == TOR, for PMP entry 57).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.81.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 56) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.81.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp56cfg[4] == 1)) {
    return pmpaddr56.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp56cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr56.ADDR & ~mask;
  } else {
    return pmpaddr56.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp56cfg[4] == 1)) {
    return pmpaddr56.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp56cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr56.ADDR & ~mask;
  } else {
    return pmpaddr56.ADDR;
  }
}
```

C.82. pmpaddr57

PMP Address 57

PMP entry address

C.82.1. Attributes

CSR Address	0x3e9
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.82.2. Format

Figure 82. pmpaddr57 format

C.82.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr57.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.82.4. Fields

pmpaddr57.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 57 (or, if `pmp58cfg.A` == TOR, for PMP entry 58).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.82.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 57) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.82.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp57cfg[4] == 1)) {
    return pmpaddr57.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp57cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr57.ADDR & ~mask;
  } else {
    return pmpaddr57.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp57cfg[4] == 1)) {
    return pmpaddr57.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp57cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr57.ADDR & ~mask;
  } else {
    return pmpaddr57.ADDR;
  }
}
```

C.83. pmpaddr58

PMP Address 58

PMP entry address

C.83.1. Attributes

CSR Address	0x3ea
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.83.2. Format

Figure 83. pmpaddr58 format

C.83.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr58.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.83.4. Fields

pmpaddr58.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 58 (or, if `pmp59cfg.A` == TOR, for PMP entry 59).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.83.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 58) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.83.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp58cfg[4] == 1)) {
    return pmpaddr58.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp58cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr58.ADDR & ~mask;
  } else {
    return pmpaddr58.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp58cfg[4] == 1)) {
    return pmpaddr58.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp58cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr58.ADDR & ~mask;
  } else {
    return pmpaddr58.ADDR;
  }
}
```

C.84. pmpaddr59

PMP Address 59

PMP entry address

C.84.1. Attributes

CSR Address	0x3eb
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.84.2. Format

Figure 84. pmpaddr59 format

C.84.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr59.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.84.4. Fields

pmpaddr59.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 59 (or, if `pmp60cfg.A` == TOR, for PMP entry 60).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.84.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 59) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.84.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp59cfg[4] == 1)) {
    return pmpaddr59.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp59cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr59.ADDR & ~mask;
  } else {
    return pmpaddr59.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp59cfg[4] == 1)) {
    return pmpaddr59.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp59cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr59.ADDR & ~mask;
  } else {
    return pmpaddr59.ADDR;
  }
}
```

C.85. pmpaddr6

PMP Address 6

PMP entry address

C.85.1. Attributes

CSR Address	0x3b6
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.85.2. Format

Figure 85. pmpaddr6 format

C.85.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr6.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.85.4. Fields

pmpaddr6.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 6 (or, if `pmp7cfg.A == TOR`, for PMP entry 7).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.85.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 6) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.85.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg1.pmp6cfg[4] == 1)) {
    return pmpaddr6.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg1.pmp6cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr6.ADDR & ~mask;
  } else {
    return pmpaddr6.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp6cfg[4] == 1)) {
    return pmpaddr6.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp6cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr6.ADDR & ~mask;
  } else {
    return pmpaddr6.ADDR;
  }
}
```

C.86. pmpaddr60

PMP Address 60

PMP entry address

C.86.1. Attributes

CSR Address	0x3ec
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.86.2. Format

Figure 86. pmpaddr60 format

C.86.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr60.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.86.4. Fields

pmpaddr60.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 60 (or, if `pmp61cfg.A` == TOR, for PMP entry 61).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.86.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 60) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.86.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg15.pmp60cfg[4] == 1)) {
    return pmpaddr60.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg15.pmp60cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr60.ADDR & ~mask;
  } else {
    return pmpaddr60.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp60cfg[4] == 1)) {
    return pmpaddr60.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp60cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr60.ADDR & ~mask;
  } else {
    return pmpaddr60.ADDR;
  }
}
```

C.87. pmpaddr61

PMP Address 61

PMP entry address

C.87.1. Attributes

CSR Address	0x3ed
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.87.2. Format

Figure 87. pmpaddr61 format

C.87.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr61.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.87.4. Fields

pmpaddr61.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 61 (or, if `pmp62cfg.A` == TOR, for PMP entry 62).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.87.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 61) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.87.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg15.pmp61cfg[4] == 1)) {
    return pmpaddr61.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg15.pmp61cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr61.ADDR & ~mask;
  } else {
    return pmpaddr61.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp61cfg[4] == 1)) {
    return pmpaddr61.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp61cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr61.ADDR & ~mask;
  } else {
    return pmpaddr61.ADDR;
  }
}
```

C.88. pmpaddr62

PMP Address 62

PMP entry address

C.88.1. Attributes

CSR Address	0x3ee
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.88.2. Format

Figure 88. pmpaddr62 format

C.88.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr62.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.88.4. Fields

pmpaddr62.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 62 (or, if `pmp63cfg.A` == TOR, for PMP entry 63).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.88.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 62) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.88.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg15.pmp62cfg[4] == 1)) {
    return pmpaddr62.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg15.pmp62cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr62.ADDR & ~mask;
  } else {
    return pmpaddr62.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp62cfg[4] == 1)) {
    return pmpaddr62.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp62cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr62.ADDR & ~mask;
  } else {
    return pmpaddr62.ADDR;
  }
}
```

C.89. pmpaddr63

PMP Address 63

PMP entry address

C.89.1. Attributes

CSR Address	0x3ef
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.89.2. Format

Figure 89. pmpaddr63 format

C.89.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr63.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.89.4. Fields

pmpaddr63.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 63 (or, if `pmp64cfg.A` == TOR, for PMP entry 64).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.89.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 63) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.89.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg15.pmp63cfg[4] == 1)) {
    return pmpaddr63.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg15.pmp63cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr63.ADDR & ~mask;
  } else {
    return pmpaddr63.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg14.pmp63cfg[4] == 1)) {
    return pmpaddr63.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg14.pmp63cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr63.ADDR & ~mask;
  } else {
    return pmpaddr63.ADDR;
  }
}
```

C.90. pmpaddr7

PMP Address 7

PMP entry address

C.90.1. Attributes

CSR Address	0x3b7
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.90.2. Format

Figure 90. pmpaddr7 format

C.90.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr7.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.90.4. Fields

pmpaddr7.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 7 (or, if `pmp8cfg.A == TOR`, for PMP entry 8).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.90.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 7) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.90.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg1.pmp7cfg[4] == 1)) {
    return pmpaddr7.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg1.pmp7cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr7.ADDR & ~mask;
  } else {
    return pmpaddr7.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg0.pmp7cfg[4] == 1)) {
    return pmpaddr7.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg0.pmp7cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr7.ADDR & ~mask;
  } else {
    return pmpaddr7.ADDR;
  }
}
```

C.91. pmpaddr8

PMP Address 8

PMP entry address

C.91.1. Attributes

CSR Address	0x3b8
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.91.2. Format

Figure 91. pmpaddr8 format

C.91.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr8.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.91.4. Fields

pmpaddr8.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 8 (or, if `pmp9cfg.A == TOR`, for PMP entry 9).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.91.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 8) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.91.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp8cfg[4] == 1)) {
    return pmpaddr8.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp8cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr8.ADDR & ~mask;
  } else {
    return pmpaddr8.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp8cfg[4] == 1)) {
    return pmpaddr8.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp8cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr8.ADDR & ~mask;
  } else {
    return pmpaddr8.ADDR;
  }
}
```

C.92. pmpaddr9

PMP Address 9

PMP entry address

C.92.1. Attributes

CSR Address	0x3b9
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.92.2. Format

Figure 92. pmpaddr9 format

C.92.3. Field Summary

Name	Location	Type	Reset Value
pmpaddr9.ADDR	63:0	RW RO	UNDEFINED_LEGAL

C.92.4. Fields

pmpaddr9.ADDR Field

Location:

63:0

Description:

Bits PHYS_ADDR_WIDTH-1:2 of the address specifier for PMP entry 9 (or, if `pmp10cfg.A` == TOR, for PMP entry 10).

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.92.5. 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:

```
ADDR = if (csr_value.ADDR >= (PHYS_ADDR_WIDTH >> 2)) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
} else if (NUM_PMP_ENTRIES > 9) {
  return UNDEFINED_LEGAL_DETERMINISTIC;
```

```
} else {
  return csr_value.ADDR;
}
```

C.92.6. Software read

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

```
if (MXLEN == 32) {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp9cfg[4] == 1)) {
    return pmpaddr9.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp9cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr9.ADDR & ~mask;
  } else {
    return pmpaddr9.ADDR;
  }
} else {
  if ((PMP_GRANULARITY >= 16) && (pmpcfg2.pmp9cfg[4] == 1)) {
    return pmpaddr9.ADDR | {PMP_GRANULARITY - 3{1'b1}};
  } else if ((PMP_GRANULARITY >= 8) && (pmpcfg2.pmp9cfg[4] == 0)) {
    Bits<PHYS_ADDR_WIDTH - 2> mask = {PMP_GRANULARITY - 2{1'b1}};
    return pmpaddr9.ADDR & ~mask;
  } else {
    return pmpaddr9.ADDR;
  }
}
```

C.93. pmpcfg0

PMP Configuration Register 0

PMP entry configuration

C.93.1. Attributes

CSR Address	0x3a0
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.93.2. Format

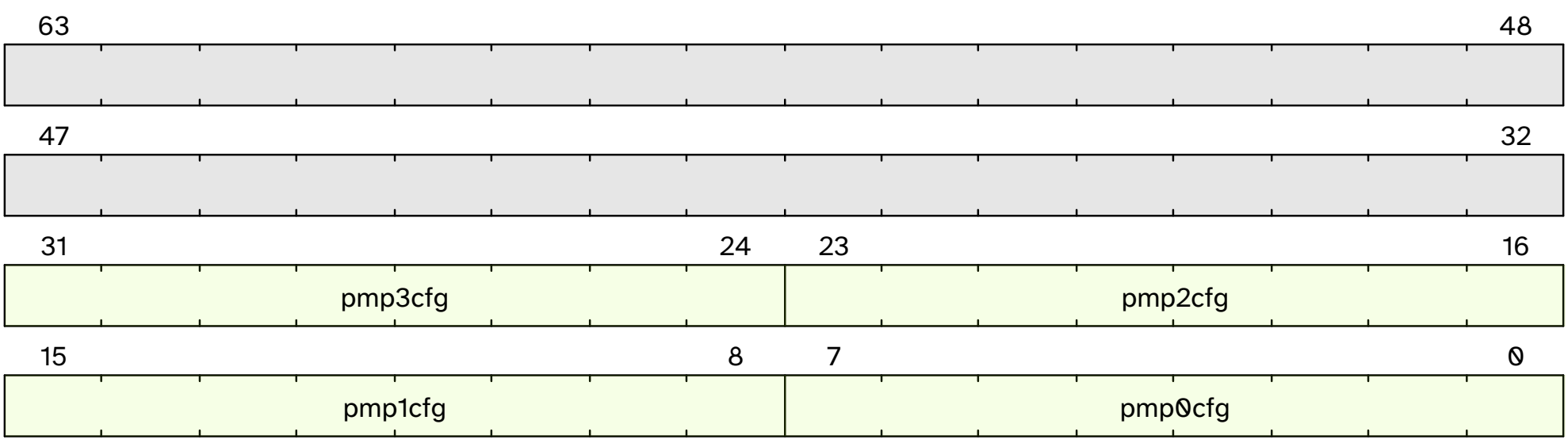


Figure 93. pmpcfg0 format

C.93.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg0.pmp0cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp1cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp2cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp3cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp4cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp5cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp6cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg0.pmp7cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.93.4. Fields

pmpcfg0.pmp0cfg Field

Location:

7:0

Description:

PMP configuration for entry 0

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp1cfg Field

Location:

15:8

Description:

PMP configuration for entry 1

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp2cfg Field

Location:

23:16

Description:

PMP configuration for entry 2

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp3cfg Field

Location:

31:24

Description:

PMP configuration for entry 3

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp4cfg Field

pmp4cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 4

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp5cfg Field

pmp5cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 5

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp6cfg Field

pmp6cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 6

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg0.pmp7cfg Field

pmp7cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 7

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.93.5. 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:

```
pmp0cfg = if ((CSR[pmpcfg0].pmp0cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp0cfg & 0x1) == 0) && ((csr_value.pmp0cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp0cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp0cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp0cfg;

pmp1cfg = if ((CSR[pmpcfg0].pmp1cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp1cfg & 0x1) == 0) && ((csr_value.pmp1cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp1cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp1cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg0].pmp1cfg;

pmp2cfg = if ((CSR[pmpcfg0].pmp2cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp2cfg & 0x1) == 0) && ((csr_value.pmp2cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp2cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp2cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp2cfg;

pmp3cfg = if ((CSR[pmpcfg0].pmp3cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp3cfg & 0x1) == 0) && ((csr_value.pmp3cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp3cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp3cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp3cfg;

pmp4cfg = if ((CSR[pmpcfg0].pmp4cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp4cfg & 0x1) == 0) && ((csr_value.pmp4cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp4cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp4cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp4cfg;

pmp5cfg = if ((xlen() == 64) && (CSR[pmpcfg0].pmp5cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp5cfg & 0x1) == 0) && ((csr_value.pmp5cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp5cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp5cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp5cfg;

pmp6cfg = if ((xlen() == 64) && (CSR[pmpcfg0].pmp6cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp6cfg & 0x1) == 0) && ((csr_value.pmp6cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp6cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp6cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp6cfg;

pmp7cfg = if ((xlen() == 64) && (CSR[pmpcfg0].pmp7cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp7cfg & 0x1) == 0) && ((csr_value.pmp7cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp7cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp7cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg0].pmp7cfg;
```

C.94. pmpcfg1

PMP Configuration Register 1

pmpcfg1 is only defined in RV32.

PMP entry configuration

C.94.1. Attributes

CSR Address	0x3a1
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.94.2. Format

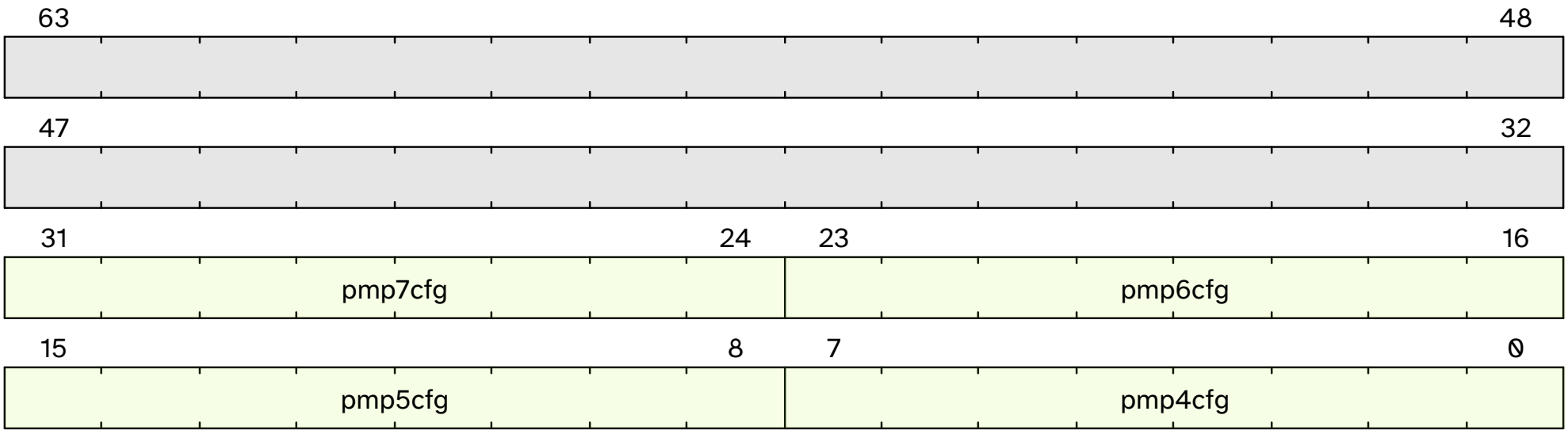


Figure 94. pmpcfg1 format

C.94.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg1.pmp4cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg1.pmp5cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg1.pmp6cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg1.pmp7cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.94.4. Fields

pmpcfg1.pmp4cfg Field

Location:

7:0

Description:

PMP configuration for entry 4

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg1.pmp5cfg Field

Location:

15:8

Description:

PMP configuration for entry 5

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg1.pmp6cfg Field

Location:

23:16

Description:

PMP configuration for entry 6

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg1.pmp7cfg Field

Location:

31:24

Description:

PMP configuration for entry 7

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

C.94.5. 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:

```
pmp4cfg = if ((CSR[pmpcfg1].pmp4cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp4cfg & 0x1) == 0) && ((csr_value.pmp4cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp4cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp4cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg1].pmp4cfg;

pmp5cfg = if ((CSR[pmpcfg1].pmp5cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp5cfg & 0x1) == 0) && ((csr_value.pmp5cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp5cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp5cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg1].pmp5cfg;

pmp6cfg = if ((CSR[pmpcfg1].pmp6cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp6cfg & 0x1) == 0) && ((csr_value.pmp6cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp6cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp6cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg1].pmp6cfg;

pmp7cfg = if ((CSR[pmpcfg1].pmp7cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp7cfg & 0x1) == 0) && ((csr_value.pmp7cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp7cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp7cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg1].pmp7cfg;
```

C.95. pmpcfg10

PMP Configuration Register 10

PMP entry configuration

C.95.1. Attributes

CSR Address	0x3aa
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.95.2. Format

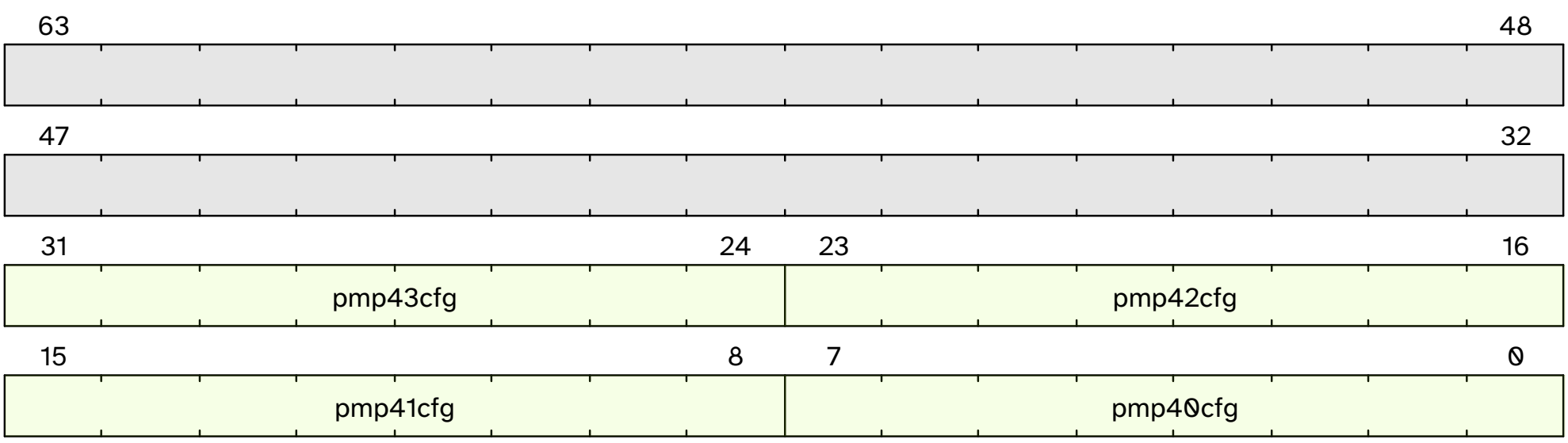


Figure 95. pmpcfg10 format

C.95.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg10.pmp40cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp41cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp42cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp43cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp44cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp45cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp46cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg10.pmp47cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.95.4. Fields

pmpcfg10.pmp40cfg Field

Location:

7:0

Description:

PMP configuration for entry 40

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp41cfg Field

Location:

15:8

Description:

PMP configuration for entry 41

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp42cfg Field

Location:

23:16

Description:

PMP configuration for entry 42

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp43cfg Field

Location:

31:24

Description:

PMP configuration for entry 43

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp44cfg Field

pmp44cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 44

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp45cfg Field

pmp45cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 45

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp46cfg Field

pmp46cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 46

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg10.pmp47cfg Field

pmp47cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 47

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.95.5. 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:

```
pmp40cfg = if ((CSR[pmpcfg10].pmp40cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp40cfg & 0x1) == 0) && ((csr_value.pmp40cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp40cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp40cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp40cfg;

pmp41cfg = if ((CSR[pmpcfg10].pmp41cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp41cfg & 0x1) == 0) && ((csr_value.pmp41cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp41cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp41cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg10].pmp41cfg;

pmp42cfg = if ((CSR[pmpcfg10].pmp42cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp42cfg & 0x1) == 0) && ((csr_value.pmp42cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp42cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp42cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp42cfg;

pmp43cfg = if ((CSR[pmpcfg10].pmp43cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp43cfg & 0x1) == 0) && ((csr_value.pmp43cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp43cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp43cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp43cfg;

pmp44cfg = if ((CSR[pmpcfg10].pmp44cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp44cfg & 0x1) == 0) && ((csr_value.pmp44cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp44cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp44cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp44cfg;

pmp45cfg = if ((xlen() == 64) && (CSR[pmpcfg10].pmp45cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp45cfg & 0x1) == 0) && ((csr_value.pmp45cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp45cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp45cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp45cfg;

pmp46cfg = if ((xlen() == 64) && (CSR[pmpcfg10].pmp46cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp46cfg & 0x1) == 0) && ((csr_value.pmp46cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp46cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp46cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp46cfg;

pmp47cfg = if ((xlen() == 64) && (CSR[pmpcfg10].pmp47cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp47cfg & 0x1) == 0) && ((csr_value.pmp47cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp47cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp47cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg10].pmp47cfg;
```

C.96. pmpcfg11

PMP Configuration Register 11

pmpcfg11 is only defined in RV32.

PMP entry configuration

C.96.1. Attributes

CSR Address	0x3ab
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.96.2. Format

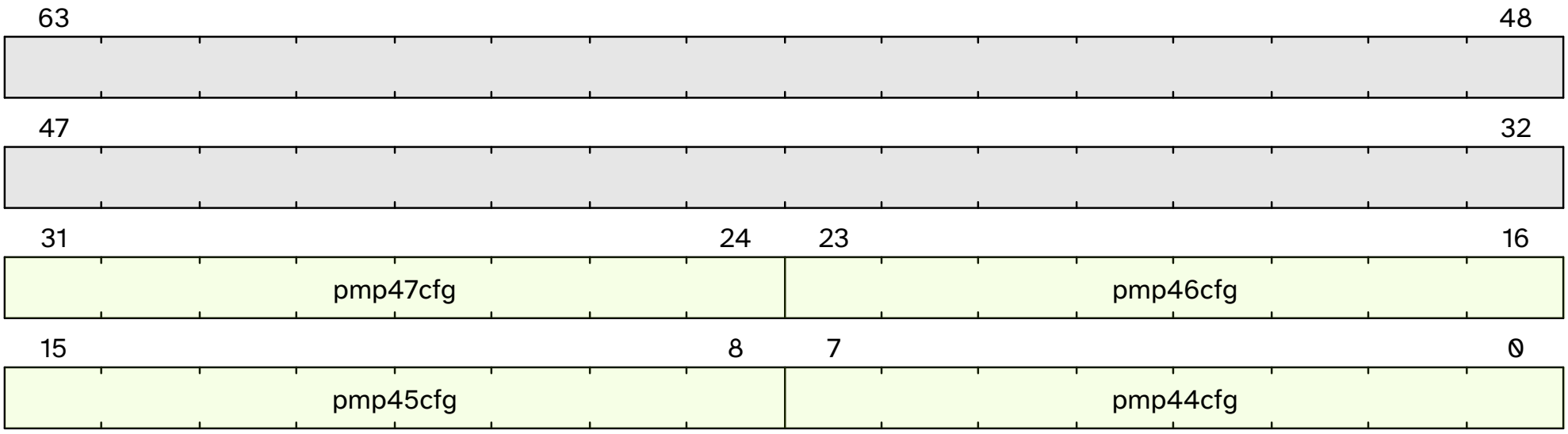


Figure 96. pmpcfg11 format

C.96.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg11.pmp44cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg11.pmp45cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg11.pmp46cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg11.pmp47cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.96.4. Fields

pmpcfg11.pmp44cfg Field

Location:

7:0

Description:

PMP configuration for entry 44

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg11.pmp45cfg Field

Location:

15:8

Description:

PMP configuration for entry 45

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg11.pmp46cfg Field

Location:

23:16

Description:

PMP configuration for entry 46

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg11.pmp47cfg Field

Location:

31:24

Description:

PMP configuration for entry 47

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

C.96.5. 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:

```
pmp44cfg = if ((CSR[pmpcfg11].pmp44cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp44cfg & 0x1) == 0) && ((csr_value.pmp44cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp44cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp44cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg11].pmp44cfg;

pmp45cfg = if ((CSR[pmpcfg11].pmp45cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp45cfg & 0x1) == 0) && ((csr_value.pmp45cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp45cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp45cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg11].pmp45cfg;

pmp46cfg = if ((CSR[pmpcfg11].pmp46cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp46cfg & 0x1) == 0) && ((csr_value.pmp46cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp46cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp46cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg11].pmp46cfg;

pmp47cfg = if ((CSR[pmpcfg11].pmp47cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp47cfg & 0x1) == 0) && ((csr_value.pmp47cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp47cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp47cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg11].pmp47cfg;
```

C.97. pmpcfg12

PMP Configuration Register 12

PMP entry configuration

C.97.1. Attributes

CSR Address	0x3ac
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.97.2. Format

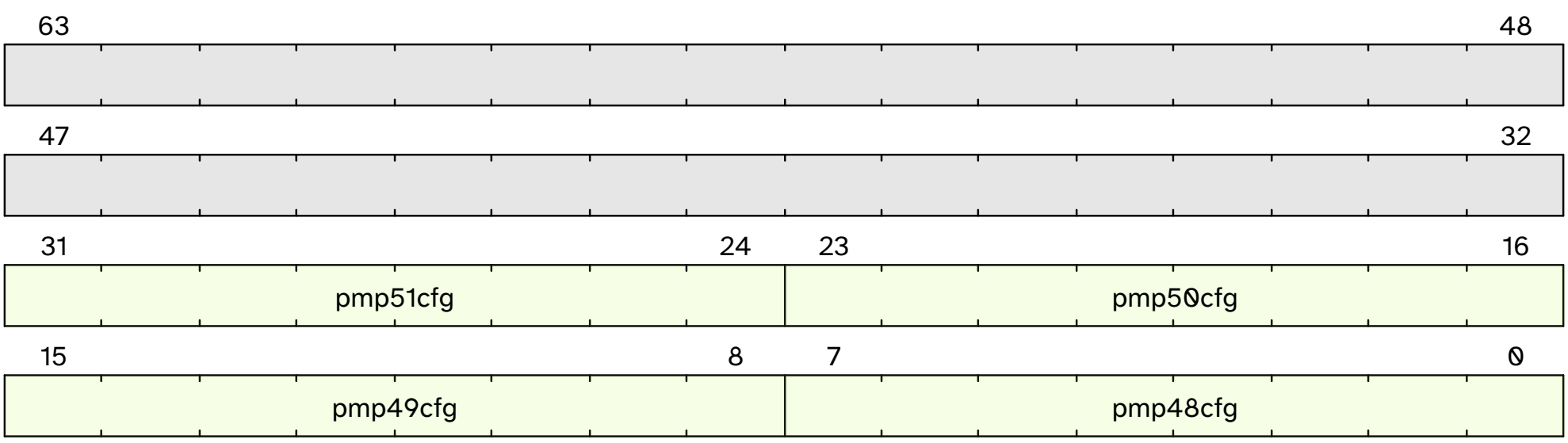


Figure 97. pmpcfg12 format

C.97.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg12.pmp48cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp49cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp50cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp51cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp52cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp53cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp54cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg12.pmp55cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.97.4. Fields

pmpcfg12.pmp48cfg Field

Location:

7:0

Description:

PMP configuration for entry 48

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp49cfg Field

Location:

15:8

Description:

PMP configuration for entry 49

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp50cfg Field

Location:

23:16

Description:

PMP configuration for entry 50

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp51cfg Field

Location:

31:24

Description:

PMP configuration for entry 51

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp52cfg Field

pmp52cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 52

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp53cfg Field

pmp53cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 53

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp54cfg Field

pmp54cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 54

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg12.pmp55cfg Field

pmp55cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 55

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.97.5. 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:

```
pmp48cfg = if ((CSR[pmpcfg12].pmp48cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp48cfg & 0x1) == 0) && ((csr_value.pmp48cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp48cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp48cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp48cfg;

pmp49cfg = if ((CSR[pmpcfg12].pmp49cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp49cfg & 0x1) == 0) && ((csr_value.pmp49cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp49cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp49cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg12].pmp49cfg;

pmp50cfg = if ((CSR[pmpcfg12].pmp50cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp50cfg & 0x1) == 0) && ((csr_value.pmp50cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp50cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp50cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp50cfg;

pmp51cfg = if ((CSR[pmpcfg12].pmp51cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp51cfg & 0x1) == 0) && ((csr_value.pmp51cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp51cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp51cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp51cfg;

pmp52cfg = if ((CSR[pmpcfg12].pmp52cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp52cfg & 0x1) == 0) && ((csr_value.pmp52cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp52cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp52cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp52cfg;

pmp53cfg = if ((xlen() == 64) && (CSR[pmpcfg12].pmp53cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp53cfg & 0x1) == 0) && ((csr_value.pmp53cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp53cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp53cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp53cfg;

pmp54cfg = if ((xlen() == 64) && (CSR[pmpcfg12].pmp54cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp54cfg & 0x1) == 0) && ((csr_value.pmp54cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp54cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp54cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp54cfg;

pmp55cfg = if ((xlen() == 64) && (CSR[pmpcfg12].pmp55cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp55cfg & 0x1) == 0) && ((csr_value.pmp55cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp55cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp55cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg12].pmp55cfg;
```

C.98. pmpcfg13

PMP Configuration Register 13

pmpcfg13 is only defined in RV32.

PMP entry configuration

C.98.1. Attributes

CSR Address	0x3ad
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.98.2. Format

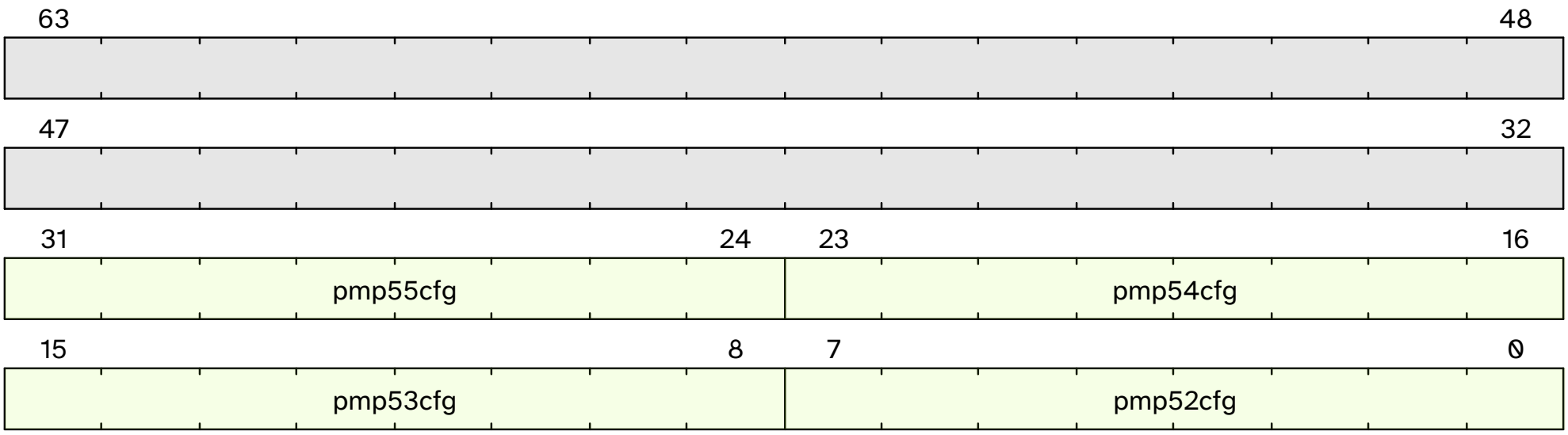


Figure 98. pmpcfg13 format

C.98.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg13.pmp52cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg13.pmp53cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg13.pmp54cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg13.pmp55cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.98.4. Fields

pmpcfg13.pmp52cfg Field

Location:

7:0

Description:

PMP configuration for entry 52

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg13.pmp53cfg Field

Location:

15:8

Description:

PMP configuration for entry 53

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg13.pmp54cfg Field

Location:

23:16

Description:

PMP configuration for entry 54

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg13.pmp55cfg Field

Location:

31:24

Description:

PMP configuration for entry 55

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

C.98.5. 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:

```
pmp52cfg = if ((CSR[pmpcfg13].pmp52cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp52cfg & 0x1) == 0) && ((csr_value.pmp52cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp52cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp52cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg13].pmp52cfg;

pmp53cfg = if ((CSR[pmpcfg13].pmp53cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp53cfg & 0x1) == 0) && ((csr_value.pmp53cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp53cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp53cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg13].pmp53cfg;

pmp54cfg = if ((CSR[pmpcfg13].pmp54cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp54cfg & 0x1) == 0) && ((csr_value.pmp54cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp54cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp54cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg13].pmp54cfg;

pmp55cfg = if ((CSR[pmpcfg13].pmp55cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp55cfg & 0x1) == 0) && ((csr_value.pmp55cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp55cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp55cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg13].pmp55cfg;
```

C.99. pmpcfg14

PMP Configuration Register 14

PMP entry configuration

C.99.1. Attributes

CSR Address	0x3ae
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.99.2. Format

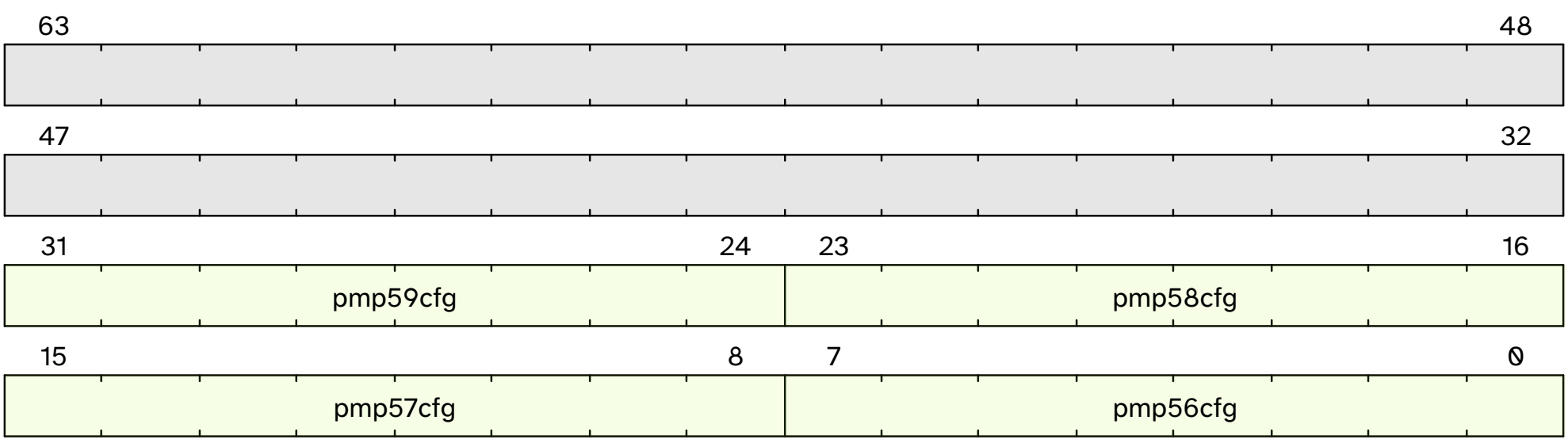


Figure 99. pmpcfg14 format

C.99.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg14.pmp56cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp57cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp58cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp59cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp60cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp61cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp62cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg14.pmp63cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.99.4. Fields

pmpcfg14.pmp56cfg Field

Location:

7:0

Description:

PMP configuration for entry 56

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp57cfg Field

Location:

15:8

Description:

PMP configuration for entry 57

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp58cfg Field

Location:

23:16

Description:

PMP configuration for entry 58

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp59cfg Field

Location:

31:24

Description:

PMP configuration for entry 59

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp60cfg Field

pmp60cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 60

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp61cfg Field

pmp61cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 61

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp62cfg Field

pmp62cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 62

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg14.pmp63cfg Field

pmp63cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 63

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.99.5. 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:

```
pmp56cfg = if ((CSR[pmpcfg14].pmp56cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp56cfg & 0x1) == 0) && ((csr_value.pmp56cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp56cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp56cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp56cfg;

pmp57cfg = if ((CSR[pmpcfg14].pmp57cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp57cfg & 0x1) == 0) && ((csr_value.pmp57cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp57cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp57cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg14].pmp57cfg;

pmp58cfg = if ((CSR[pmpcfg14].pmp58cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp58cfg & 0x1) == 0) && ((csr_value.pmp58cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp58cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp58cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp58cfg;

pmp59cfg = if ((CSR[pmpcfg14].pmp59cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp59cfg & 0x1) == 0) && ((csr_value.pmp59cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp59cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp59cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp59cfg;

pmp60cfg = if ((CSR[pmpcfg14].pmp60cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp60cfg & 0x1) == 0) && ((csr_value.pmp60cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp60cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp60cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp60cfg;

pmp61cfg = if ((xlen() == 64) && (CSR[pmpcfg14].pmp61cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp61cfg & 0x1) == 0) && ((csr_value.pmp61cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp61cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp61cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp61cfg;

pmp62cfg = if ((xlen() == 64) && (CSR[pmpcfg14].pmp62cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp62cfg & 0x1) == 0) && ((csr_value.pmp62cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp62cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp62cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp62cfg;

pmp63cfg = if ((xlen() == 64) && (CSR[pmpcfg14].pmp63cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp63cfg & 0x1) == 0) && ((csr_value.pmp63cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp63cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp63cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg14].pmp63cfg;
```

C.100. pmpcfg15

PMP Configuration Register 15

pmpcfg15 is only defined in RV32.

PMP entry configuration

C.100.1. Attributes

CSR Address	0x3af
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.100.2. Format

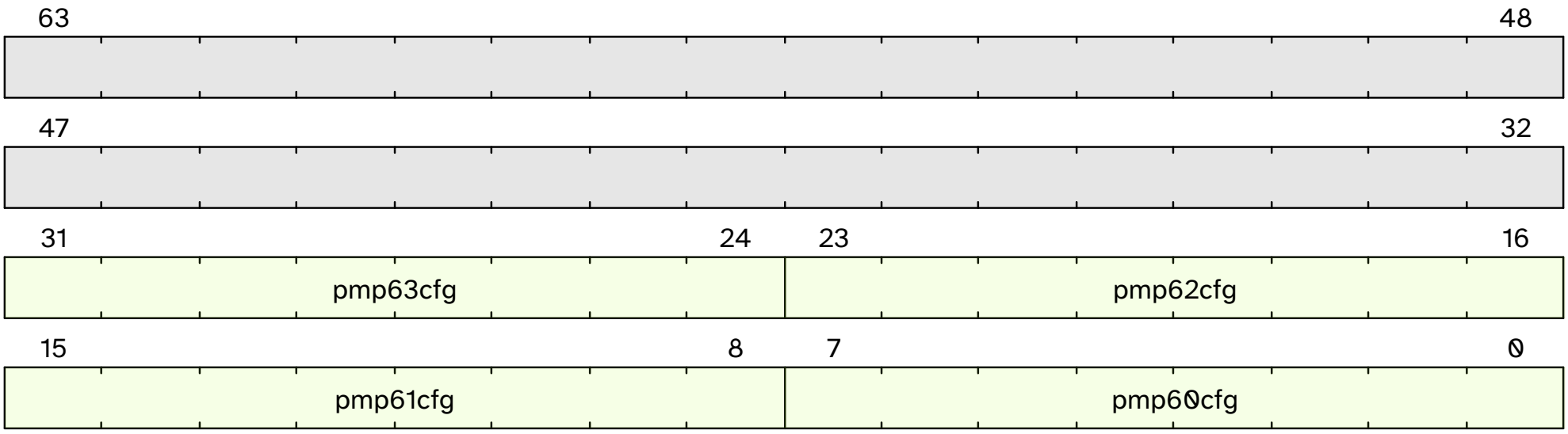


Figure 100. pmpcfg15 format

C.100.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg15.pmp60cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg15.pmp61cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg15.pmp62cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg15.pmp63cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.100.4. Fields

pmpcfg15.pmp60cfg Field

Location:

7:0

Description:

PMP configuration for entry 60

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg15.pmp61cfg Field

Location:

15:8

Description:

PMP configuration for entry 61

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg15.pmp62cfg Field

Location:

23:16

Description:

PMP configuration for entry 62

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg15.pmp63cfg Field

Location:

31:24

Description:

PMP configuration for entry 63

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.100.5. 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:

```
pmp60cfg = if ((CSR[pmpcfg15].pmp60cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp60cfg & 0x1) == 0) && ((csr_value.pmp60cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp60cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp60cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg15].pmp60cfg;

pmp61cfg = if ((CSR[pmpcfg15].pmp61cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp61cfg & 0x1) == 0) && ((csr_value.pmp61cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp61cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp61cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg15].pmp61cfg;

pmp62cfg = if ((CSR[pmpcfg15].pmp62cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp62cfg & 0x1) == 0) && ((csr_value.pmp62cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp62cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp62cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg15].pmp62cfg;

pmp63cfg = if ((CSR[pmpcfg15].pmp63cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp63cfg & 0x1) == 0) && ((csr_value.pmp63cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp63cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp63cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg15].pmp63cfg;
```

C.101. pmpcfg2

PMP Configuration Register 2

PMP entry configuration

C.101.1. Attributes

CSR Address	0x3a2
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.101.2. Format

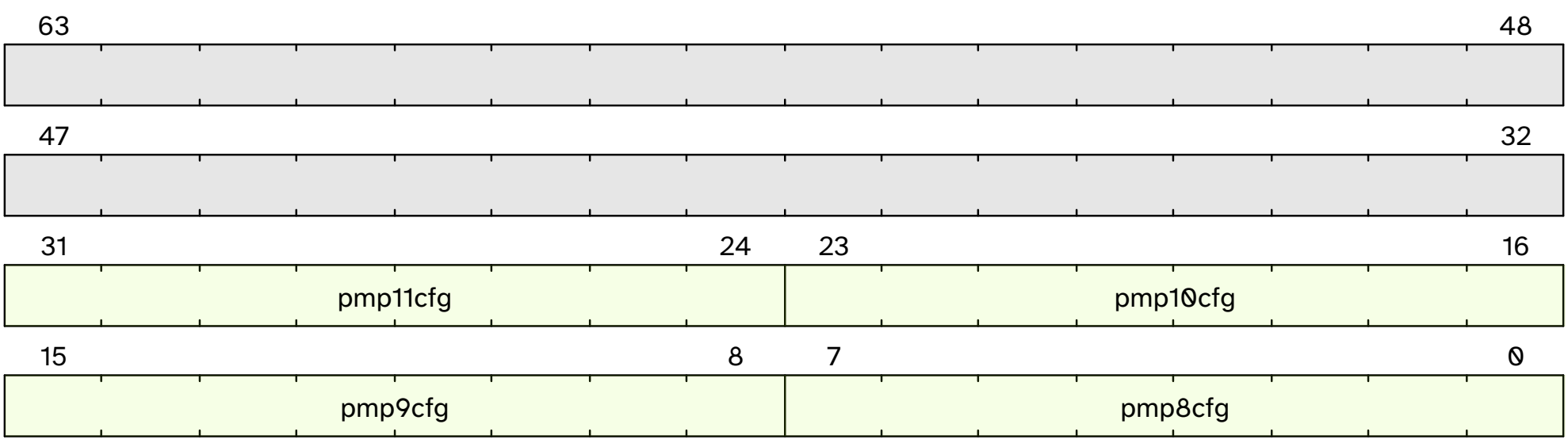


Figure 101. pmpcfg2 format

C.101.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg2.pmp8cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp9cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp10cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp11cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp12cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp13cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp14cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg2.pmp15cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.101.4. Fields

pmpcfg2.pmp8cfg Field

Location:

7:0

Description:

PMP configuration for entry 8

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp9cfg Field

Location:

15:8

Description:

PMP configuration for entry 9

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp10cfg Field

Location:

23:16

Description:

PMP configuration for entry 10

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp11cfg Field

Location:

31:24

Description:

PMP configuration for entry 11

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp12cfg Field

pmp12cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 12

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp13cfg Field

pmp13cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 13

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp14cfg Field

pmp14cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 14

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg2.pmp15cfg Field

pmp15cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 15

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.101.5. 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:

```
pmp8cfg = if ((CSR[pmpcfg2].pmp8cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp8cfg & 0x1) == 0) && ((csr_value.pmp8cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp8cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp8cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp8cfg;

pmp9cfg = if ((CSR[pmpcfg2].pmp9cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp9cfg & 0x1) == 0) && ((csr_value.pmp9cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp9cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp9cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg2].pmp9cfg;

pmp10cfg = if ((CSR[pmpcfg2].pmp10cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp10cfg & 0x1) == 0) && ((csr_value.pmp10cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp10cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp10cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp10cfg;

pmp11cfg = if ((CSR[pmpcfg2].pmp11cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp11cfg & 0x1) == 0) && ((csr_value.pmp11cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp11cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp11cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp11cfg;

pmp12cfg = if ((CSR[pmpcfg2].pmp12cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp12cfg & 0x1) == 0) && ((csr_value.pmp12cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp12cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp12cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp12cfg;

pmp13cfg = if ((xlen() == 64) && (CSR[pmpcfg2].pmp13cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp13cfg & 0x1) == 0) && ((csr_value.pmp13cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp13cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp13cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp13cfg;

pmp14cfg = if ((xlen() == 64) && (CSR[pmpcfg2].pmp14cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp14cfg & 0x1) == 0) && ((csr_value.pmp14cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp14cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp14cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp14cfg;

pmp15cfg = if ((xlen() == 64) && (CSR[pmpcfg2].pmp15cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp15cfg & 0x1) == 0) && ((csr_value.pmp15cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp15cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp15cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg2].pmp15cfg;
```

C.102. pmpcfg3

PMP Configuration Register 3

pmpcfg3 is only defined in RV32.

PMP entry configuration

C.102.1. Attributes

CSR Address	0x3a3
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.102.2. Format

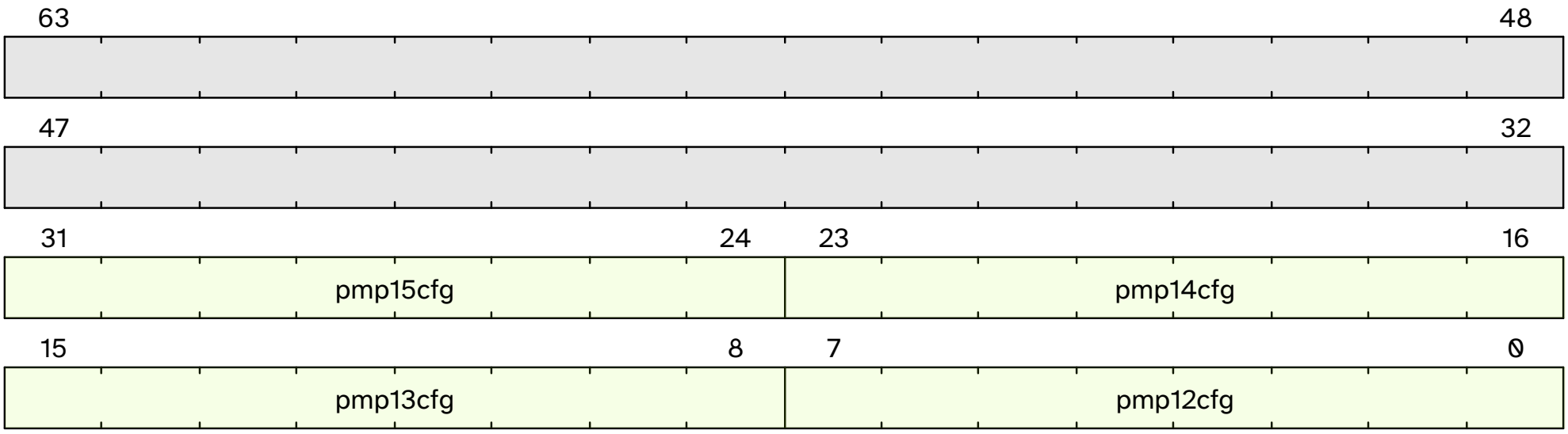


Figure 102. pmpcfg3 format

C.102.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg3.pmp12cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg3.pmp13cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg3.pmp14cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg3.pmp15cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.102.4. Fields

pmpcfg3.pmp12cfg Field

Location:

7:0

Description:

PMP configuration for entry 12

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg3.pmp13cfg Field

Location:

15:8

Description:

PMP configuration for entry 13

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg3.pmp14cfg Field

Location:

23:16

Description:

PMP configuration for entry 14

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg3.pmp15cfg Field

Location:

31:24

Description:

PMP configuration for entry 15

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

C.102.5. 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:

```
pmp12cfg = if ((CSR[pmpcfg3].pmp12cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp12cfg & 0x1) == 0) && ((csr_value.pmp12cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp12cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp12cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg3].pmp12cfg;

pmp13cfg = if ((CSR[pmpcfg3].pmp13cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp13cfg & 0x1) == 0) && ((csr_value.pmp13cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp13cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp13cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg3].pmp13cfg;

pmp14cfg = if ((CSR[pmpcfg3].pmp14cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp14cfg & 0x1) == 0) && ((csr_value.pmp14cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp14cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp14cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg3].pmp14cfg;

pmp15cfg = if ((CSR[pmpcfg3].pmp15cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp15cfg & 0x1) == 0) && ((csr_value.pmp15cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp15cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp15cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg3].pmp15cfg;
```

C.103. pmpcfg4

PMP Configuration Register 4

PMP entry configuration

C.103.1. Attributes

CSR Address	0x3a4
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.103.2. Format

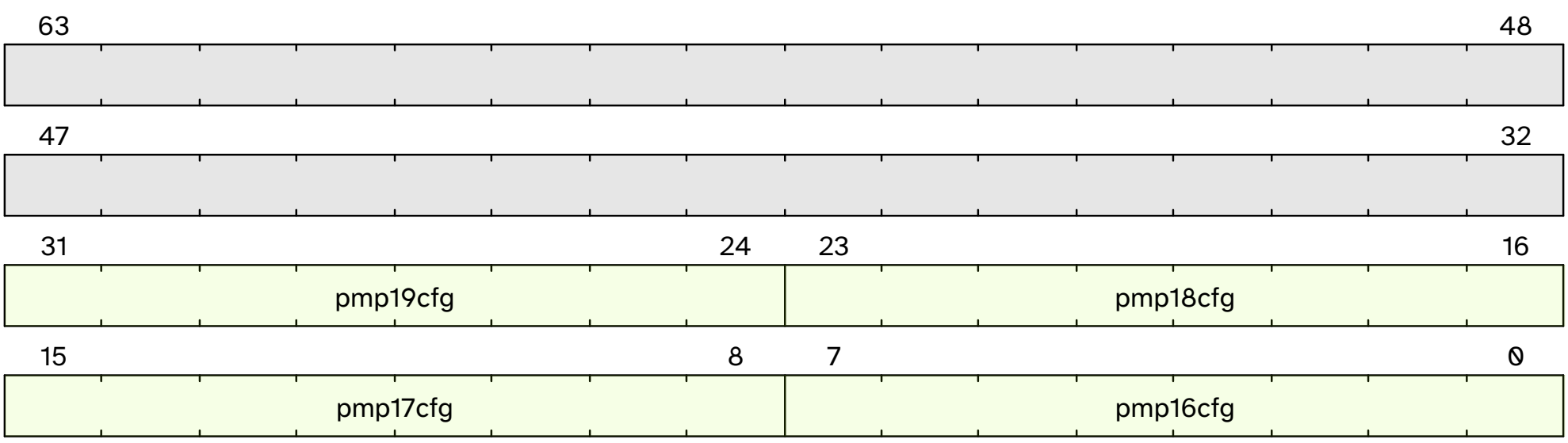


Figure 103. pmpcfg4 format

C.103.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg4.pmp16cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp17cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp18cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp19cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp20cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp21cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp22cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg4.pmp23cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.103.4. Fields

pmpcfg4.pmp16cfg Field

Location:

7:0

Description:

PMP configuration for entry 16

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp17cfg Field

Location:

15:8

Description:

PMP configuration for entry 17

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp18cfg Field

Location:

23:16

Description:

PMP configuration for entry 18

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp19cfg Field

Location:

31:24

Description:

PMP configuration for entry 19

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp20cfg Field

pmp20cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 20

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp21cfg Field

pmp21cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 21

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp22cfg Field

pmp22cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 22

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg4.pmp23cfg Field

pmp23cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 23

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.103.5. 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:

```
pmp16cfg = if ((CSR[pmpcfg4].pmp16cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp16cfg & 0x1) == 0) && ((csr_value.pmp16cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp16cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp16cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp16cfg;

pmp17cfg = if ((CSR[pmpcfg4].pmp17cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp17cfg & 0x1) == 0) && ((csr_value.pmp17cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp17cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp17cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg4].pmp17cfg;

pmp18cfg = if ((CSR[pmpcfg4].pmp18cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp18cfg & 0x1) == 0) && ((csr_value.pmp18cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp18cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp18cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp18cfg;

pmp19cfg = if ((CSR[pmpcfg4].pmp19cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp19cfg & 0x1) == 0) && ((csr_value.pmp19cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp19cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp19cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp19cfg;

pmp20cfg = if ((CSR[pmpcfg4].pmp20cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp20cfg & 0x1) == 0) && ((csr_value.pmp20cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp20cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp20cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp20cfg;

pmp21cfg = if ((xlen() == 64) && (CSR[pmpcfg4].pmp21cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp21cfg & 0x1) == 0) && ((csr_value.pmp21cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp21cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp21cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp21cfg;

pmp22cfg = if ((xlen() == 64) && (CSR[pmpcfg4].pmp22cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp22cfg & 0x1) == 0) && ((csr_value.pmp22cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp22cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp22cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp22cfg;

pmp23cfg = if ((xlen() == 64) && (CSR[pmpcfg4].pmp23cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp23cfg & 0x1) == 0) && ((csr_value.pmp23cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp23cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp23cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg4].pmp23cfg;
```

C.104. pmpcfg5

PMP Configuration Register 5

pmpcfg5 is only defined in RV32.

PMP entry configuration

C.104.1. Attributes

CSR Address	0x3a5
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.104.2. Format

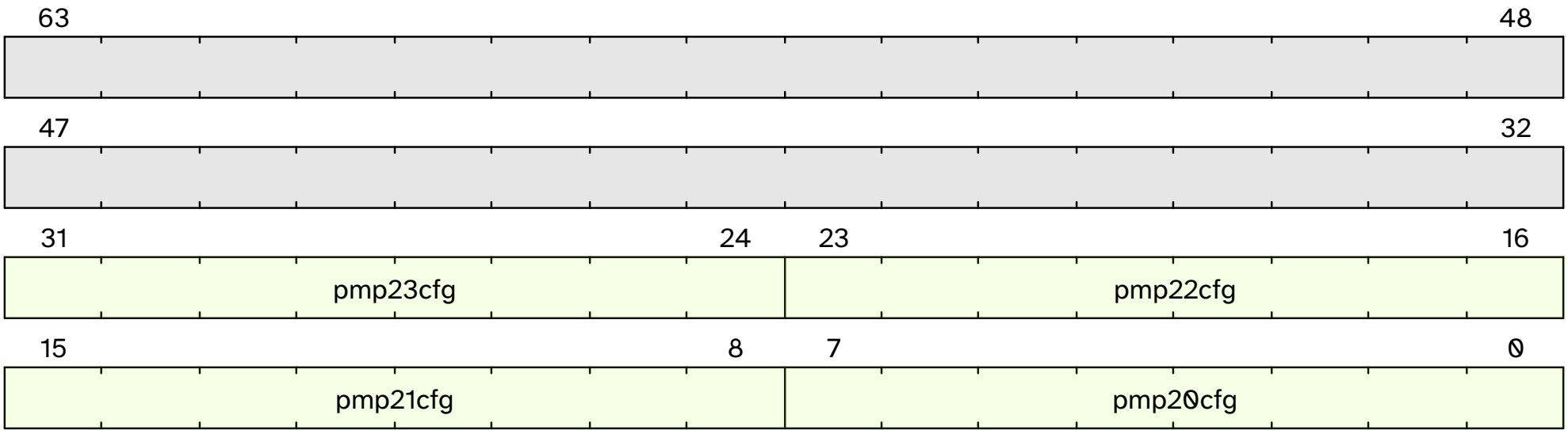


Figure 104. pmpcfg5 format

C.104.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg5.pmp20cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg5.pmp21cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg5.pmp22cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg5.pmp23cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.104.4. Fields

pmpcfg5.pmp20cfg Field

Location:

7:0

Description:

PMP configuration for entry 20

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg5.pmp21cfg Field

Location:

15:8

Description:

PMP configuration for entry 21

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg5.pmp22cfg Field

Location:

23:16

Description:

PMP configuration for entry 22

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg5.pmp23cfg Field

Location:

31:24

Description:

PMP configuration for entry 23

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.104.5. 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:

```
pmp20cfg = if ((CSR[pmpcfg5].pmp20cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp20cfg & 0x1) == 0) && ((csr_value.pmp20cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp20cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp20cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg5].pmp20cfg;

pmp21cfg = if ((CSR[pmpcfg5].pmp21cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp21cfg & 0x1) == 0) && ((csr_value.pmp21cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp21cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp21cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg5].pmp21cfg;

pmp22cfg = if ((CSR[pmpcfg5].pmp22cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp22cfg & 0x1) == 0) && ((csr_value.pmp22cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp22cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp22cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg5].pmp22cfg;

pmp23cfg = if ((CSR[pmpcfg5].pmp23cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp23cfg & 0x1) == 0) && ((csr_value.pmp23cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp23cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp23cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg5].pmp23cfg;
```

C.105. pmpcfg6

PMP Configuration Register 6

PMP entry configuration

C.105.1. Attributes

CSR Address	0x3a6
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.105.2. Format

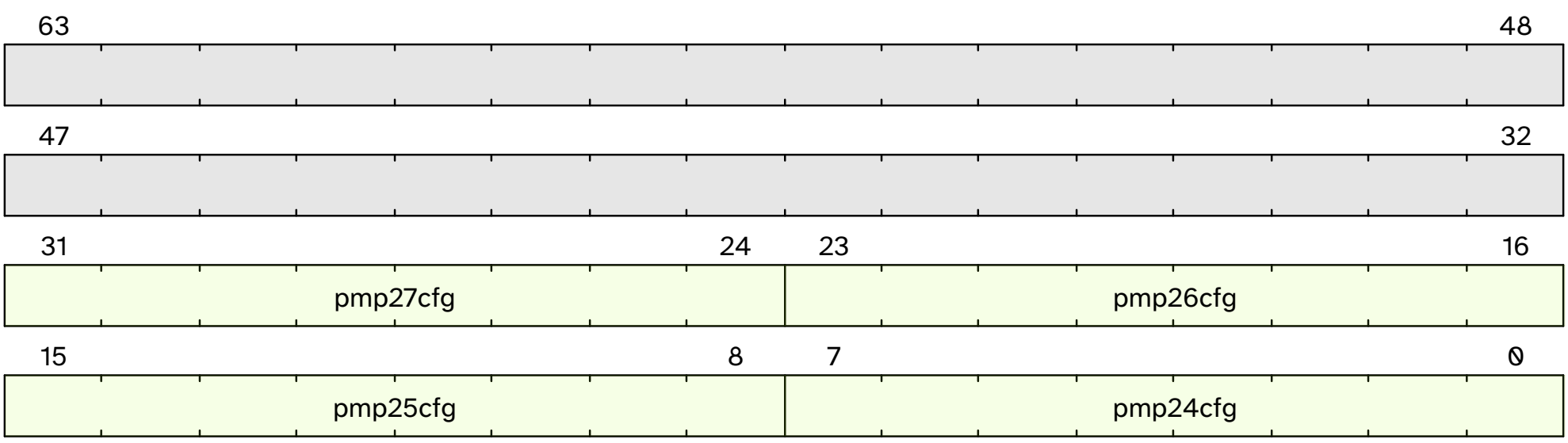


Figure 105. pmpcfg6 format

C.105.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg6.pmp24cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp25cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp26cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp27cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp28cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp29cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp30cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg6.pmp31cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.105.4. Fields

pmpcfg6.pmp24cfg Field

Location:

7:0

Description:

PMP configuration for entry 24

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp25cfg Field

Location:

15:8

Description:

PMP configuration for entry 25

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp26cfg Field

Location:

23:16

Description:

PMP configuration for entry 26

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp27cfg Field

Location:

31:24

Description:

PMP configuration for entry 27

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp28cfg Field

pmp28cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 28

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp29cfg Field

pmp29cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 29

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp30cfg Field

pmp30cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 30

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg6.pmp31cfg Field

pmp31cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 31

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.105.5. 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:

```
pmp24cfg = if ((CSR[pmpcfg6].pmp24cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp24cfg & 0x1) == 0) && ((csr_value.pmp24cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp24cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp24cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp24cfg;

pmp25cfg = if ((CSR[pmpcfg6].pmp25cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp25cfg & 0x1) == 0) && ((csr_value.pmp25cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp25cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp25cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg6].pmp25cfg;

pmp26cfg = if ((CSR[pmpcfg6].pmp26cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp26cfg & 0x1) == 0) && ((csr_value.pmp26cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp26cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp26cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp26cfg;

pmp27cfg = if ((CSR[pmpcfg6].pmp27cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp27cfg & 0x1) == 0) && ((csr_value.pmp27cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp27cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp27cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp27cfg;

pmp28cfg = if ((CSR[pmpcfg6].pmp28cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp28cfg & 0x1) == 0) && ((csr_value.pmp28cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp28cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp28cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp28cfg;

pmp29cfg = if ((xlen() == 64) && (CSR[pmpcfg6].pmp29cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp29cfg & 0x1) == 0) && ((csr_value.pmp29cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp29cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp29cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp29cfg;

pmp30cfg = if ((xlen() == 64) && (CSR[pmpcfg6].pmp30cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp30cfg & 0x1) == 0) && ((csr_value.pmp30cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp30cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp30cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp30cfg;

pmp31cfg = if ((xlen() == 64) && (CSR[pmpcfg6].pmp31cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp31cfg & 0x1) == 0) && ((csr_value.pmp31cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp31cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp31cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg6].pmp31cfg;
```

C.106. pmpcfg7

PMP Configuration Register 7

pmpcfg7 is only defined in RV32.

PMP entry configuration

C.106.1. Attributes

CSR Address	0x3a7
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.106.2. Format

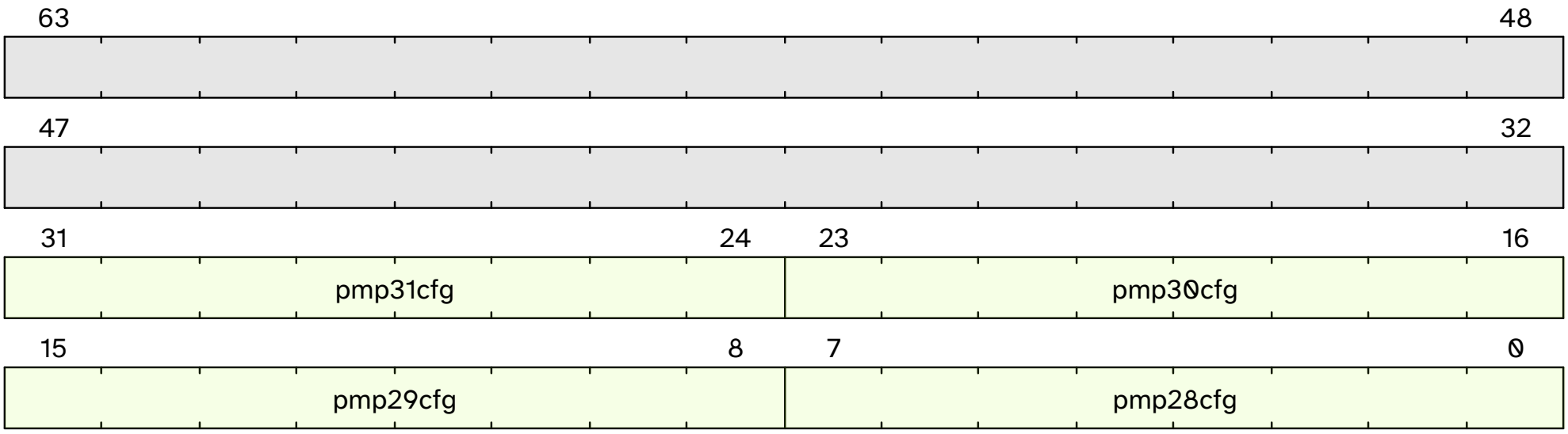


Figure 106. pmpcfg7 format

C.106.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg7.pmp28cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg7.pmp29cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg7.pmp30cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg7.pmp31cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.106.4. Fields

pmpcfg7.pmp28cfg Field

Location:

7:0

Description:

PMP configuration for entry 28

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg7.pmp29cfg Field

Location:

15:8

Description:

PMP configuration for entry 29

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg7.pmp30cfg Field

Location:

23:16

Description:

PMP configuration for entry 30

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg7.pmp31cfg Field

Location:

31:24

Description:

PMP configuration for entry 31

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

C.106.5. 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:

```
pmp28cfg = if ((CSR[pmpcfg7].pmp28cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp28cfg & 0x1) == 0) && ((csr_value.pmp28cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp28cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp28cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg7].pmp28cfg;

pmp29cfg = if ((CSR[pmpcfg7].pmp29cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp29cfg & 0x1) == 0) && ((csr_value.pmp29cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp29cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp29cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg7].pmp29cfg;

pmp30cfg = if ((CSR[pmpcfg7].pmp30cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp30cfg & 0x1) == 0) && ((csr_value.pmp30cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp30cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp30cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg7].pmp30cfg;

pmp31cfg = if ((CSR[pmpcfg7].pmp31cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp31cfg & 0x1) == 0) && ((csr_value.pmp31cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp31cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp31cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg7].pmp31cfg;
```

C.107. pmpcfg8

PMP Configuration Register 8

PMP entry configuration

C.107.1. Attributes

CSR Address	0x3a8
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.107.2. Format

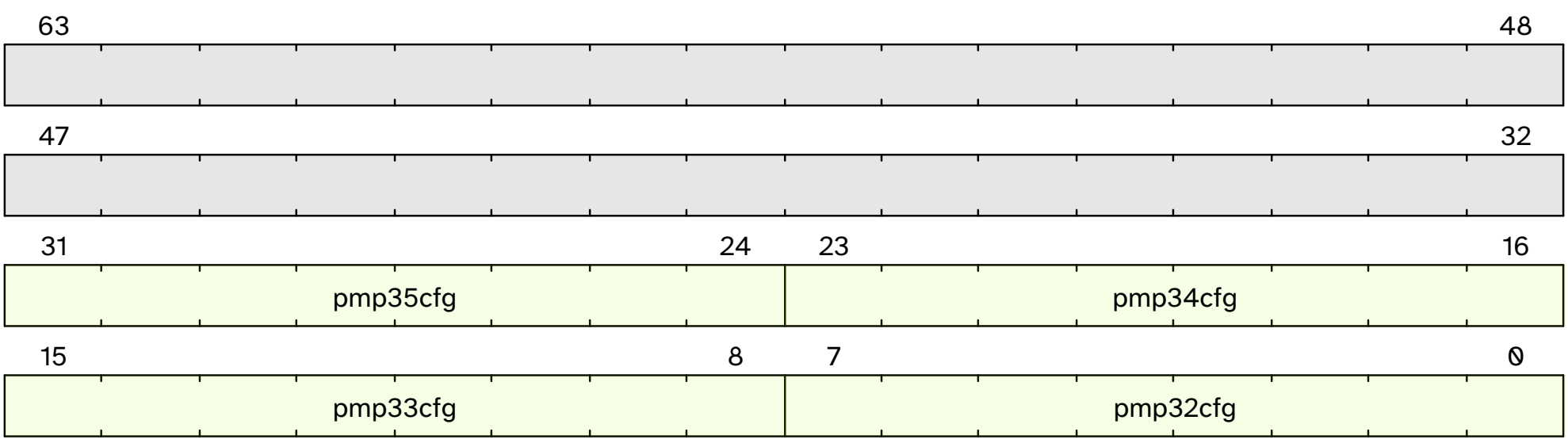


Figure 107. pmpcfg8 format

C.107.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg8.pmp32cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp33cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp34cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp35cfg	31:24	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp36cfg	39:32	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp37cfg	47:40	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp38cfg	55:48	RW-R RO	UNDEFINED_LEGAL
pmpcfg8.pmp39cfg	63:56	RW-R RO	UNDEFINED_LEGAL

C.107.4. Fields

pmpcfg8.pmp32cfg Field

Location:

7:0

Description:

PMP configuration for entry 32

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp33cfg Field

Location:

15:8

Description:

PMP configuration for entry 33

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp34cfg Field

Location:

23:16

Description:

PMP configuration for entry 34

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp35cfg Field

Location:

31:24

Description:

PMP configuration for entry 35

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp36cfg Field

pmp36cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

39:32

Description:

PMP configuration for entry 36

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 39 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 38:37 ! Reserved Writes shall be ignored.
h! A ! 36:35
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 34 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 33 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 32 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp37cfg Field

pmp37cfg is only defined in RV64 (`CSR[misa].MXL == 1`*)*

Location:

47:40

Description:

PMP configuration for entry 37

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 47 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 46:45 ! Reserved Writes shall be ignored.
h! A ! 44:43
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 42 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 41 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 40 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp38cfg Field

pmp38cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

55:48

Description:

PMP configuration for entry 38

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 55 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 54:53 ! Reserved Writes shall be ignored.
h! A ! 52:51
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 50 ! When clear, instruction fetches cause an `Access Fault` for the matching region and privilege mode.
  h! W ! 49 ! When clear, stores and AMOs cause an `Access Fault` for the matching region and privilege mode.
  h! R ! 48 ! When clear, loads cause an `Access Fault` for the matching region and privilege mode.
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg8.pmp39cfg Field

pmp39cfg is only defined in RV64 (`CSR[misa].MXL == 1`)

Location:

63:56

Description:

PMP configuration for entry 39

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 63 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 62:61 ! Reserved Writes shall be ignored.
h! A ! 60:59
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 58 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 57 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 56 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.107.5. 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:

```
pmp32cfg = if ((CSR[pmpcfg8].pmp32cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp32cfg & 0x1) == 0) && ((csr_value.pmp32cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp32cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp32cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp32cfg;

pmp33cfg = if ((CSR[pmpcfg8].pmp33cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp33cfg & 0x1) == 0) && ((csr_value.pmp33cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp33cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp33cfg;
    }
  }
}
```

```
# fall through: keep old value
return CSR[pmpcfg8].pmp33cfg;

pmp34cfg = if ((CSR[pmpcfg8].pmp34cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp34cfg & 0x1) == 0) && ((csr_value.pmp34cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp34cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp34cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp34cfg;

pmp35cfg = if ((CSR[pmpcfg8].pmp35cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp35cfg & 0x1) == 0) && ((csr_value.pmp35cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp35cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp35cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp35cfg;

pmp36cfg = if ((CSR[pmpcfg8].pmp36cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp36cfg & 0x1) == 0) && ((csr_value.pmp36cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp36cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp36cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp36cfg;

pmp37cfg = if ((xlen() == 64) && (CSR[pmpcfg8].pmp37cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp37cfg & 0x1) == 0) && ((csr_value.pmp37cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp37cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp37cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp37cfg;

pmp38cfg = if ((xlen() == 64) && (CSR[pmpcfg8].pmp38cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp38cfg & 0x1) == 0) && ((csr_value.pmp38cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp38cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp38cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp38cfg;

pmp39cfg = if ((xlen() == 64) && (CSR[pmpcfg8].pmp39cfg & 0x80) == 0) {
  # entry is not locked
```

```
  if (!(((csr_value.pmp39cfg & 0x1) == 0) && ((csr_value.pmp39cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp39cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp39cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg8].pmp39cfg;
```

C.108. pmpcfg9

PMP Configuration Register 9

pmpcfg9 is only defined in RV32.

PMP entry configuration

C.108.1. Attributes

CSR Address	0x3a9
Defining extension	• Smpmp, version >= Smpmp@1.11.0
Length	64-bit
Privilege Mode	M

C.108.2. Format

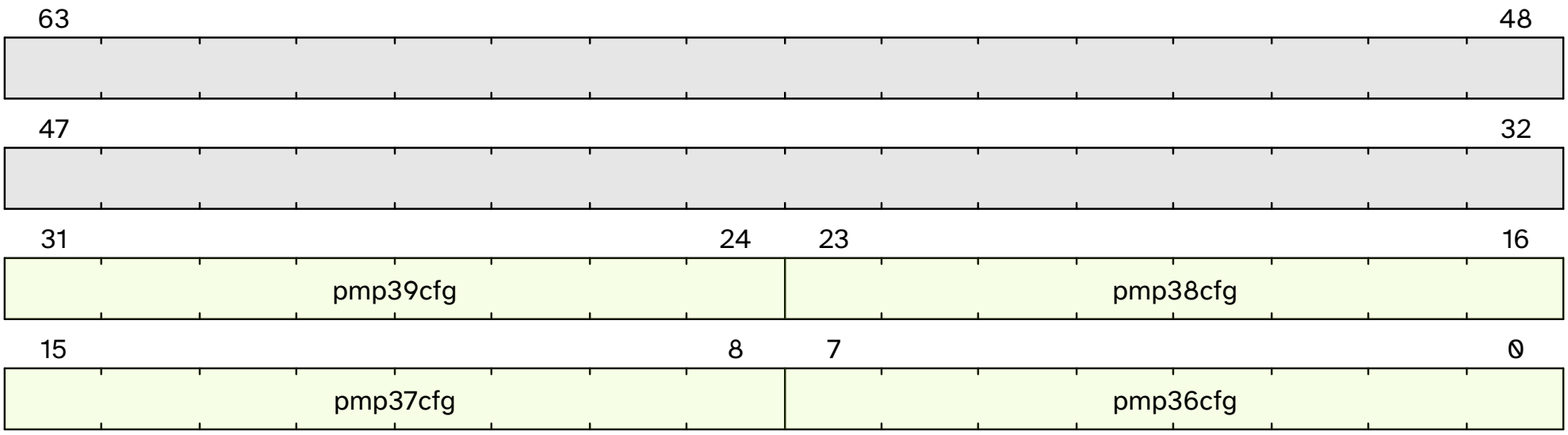


Figure 108. pmpcfg9 format

C.108.3. Field Summary

Name	Location	Type	Reset Value
pmpcfg9.pmp36cfg	7:0	RW-R RO	UNDEFINED_LEGAL
pmpcfg9.pmp37cfg	15:8	RW-R RO	UNDEFINED_LEGAL
pmpcfg9.pmp38cfg	23:16	RW-R RO	UNDEFINED_LEGAL
pmpcfg9.pmp39cfg	31:24	RW-R RO	UNDEFINED_LEGAL

C.108.4. Fields

pmpcfg9.pmp36cfg Field

Location:

7:0

Description:

PMP configuration for entry 36

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 7 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 6:5 ! Reserved Writes shall be ignored.
h! A ! 4:3

a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 2 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 1 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 0 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg9.pmp37cfg Field

Location:

15:8

Description:

PMP configuration for entry 37

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 15 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 14:13 ! Reserved Writes shall be ignored.
h! A ! 12:11
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 10 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 9 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 8 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg9.pmp38cfg Field

Location:

23:16

Description:

PMP configuration for entry 38

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 23 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 22:21 ! Reserved Writes shall be ignored.
h! A ! 20:19
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 18 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 17 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 16 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:

UNDEFINED_LEGAL

pmpcfg9.pmp39cfg Field

Location:

31:24

Description:

PMP configuration for entry 39

The bits are as follows:

[separator="!",%autowidth]
!===
! Name ! Location ! Description

h! L ! 31 ! Locks the entry from further modification. Additionally, when set, PMP checks also apply to M-mode for the entry.
h! - ! 30:29 ! Reserved Writes shall be ignored.
h! A ! 28:27
a! Address matching mode. One of:

[when="PMP_GRANULARITY < 2"]

- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NA4** *(2) - Naturally aligned four-byte region*
- **NAPOT** *(3) - Naturally aligned power of two*
  + [when="PMP_GRANULARITY >= 2"]
- **OFF** *(0) - Null region (disabled)*
- **TOR** *(1) - Top of range*
- **NAPOT** *(3) - Naturally aligned power of two*

  [when="PMP_GRANULARITY >= 2"]
  Naturally aligned four-byte region, **NA4** *(2), is not valid (not needed when the PMP granularity is larger than 4 bytes).*

  h! X ! 26 ! When clear, instruction fetches cause an `Access Fault` *for the matching region and privilege mode.*
  h! W ! 25 ! When clear, stores and AMOs cause an `Access Fault` *for the matching region and privilege mode.*
  h! R ! 24 ! When clear, loads cause an `Access Fault` *for the matching region and privilege mode.*
  !===

  The combination of R = 0, W = 1 is reserved.

Type:

RW-R

RO

Reset value:
UNDEFINED_LEGAL

C.108.5. 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:

```
pmp36cfg = if ((CSR[pmpcfg9].pmp36cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp36cfg & 0x1) == 0) && ((csr_value.pmp36cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp36cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp36cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg9].pmp36cfg;

pmp37cfg = if ((CSR[pmpcfg9].pmp37cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp37cfg & 0x1) == 0) && ((csr_value.pmp37cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp37cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp37cfg;
    }
  }
}
# fall through: keep old value
```

```
return CSR[pmpcfg9].pmp37cfg;

pmp38cfg = if ((CSR[pmpcfg9].pmp38cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp38cfg & 0x1) == 0) && ((csr_value.pmp38cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp38cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp38cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg9].pmp38cfg;

pmp39cfg = if ((CSR[pmpcfg9].pmp39cfg & 0x80) == 0) {
  # entry is not locked
  if (!(((csr_value.pmp39cfg & 0x1) == 0) && ((csr_value.pmp39cfg & 0x2) == 0x2))) {
    # not R = 0, W =1, which is reserved
    if ((PMP_GRANULARITY < 2) ||
        ((csr_value.pmp39cfg & 0x18) != 0x10)) {
      # NA4 is not allowed when PMP granularity is larger than 4 bytes
      return csr_value.pmp39cfg;
    }
  }
}
# fall through: keep old value
return CSR[pmpcfg9].pmp39cfg;
```

C.109. satp

Supervisor Address Translation and Protection

Controls the translation mode in (H)S-mode and U-mode, and holds the current ASID and page table base pointer.

C.109.1. Attributes

CSR Address	0x180
Defining extension	• S, version >= S@1.11.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.109.2. Format

This CSR format changes dynamically with XLEN.

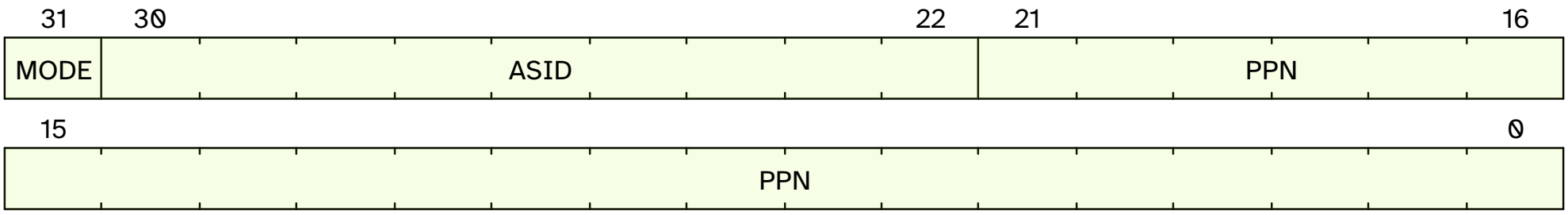


Figure 109. satp Format when CSR[mstatus].SXL == 0

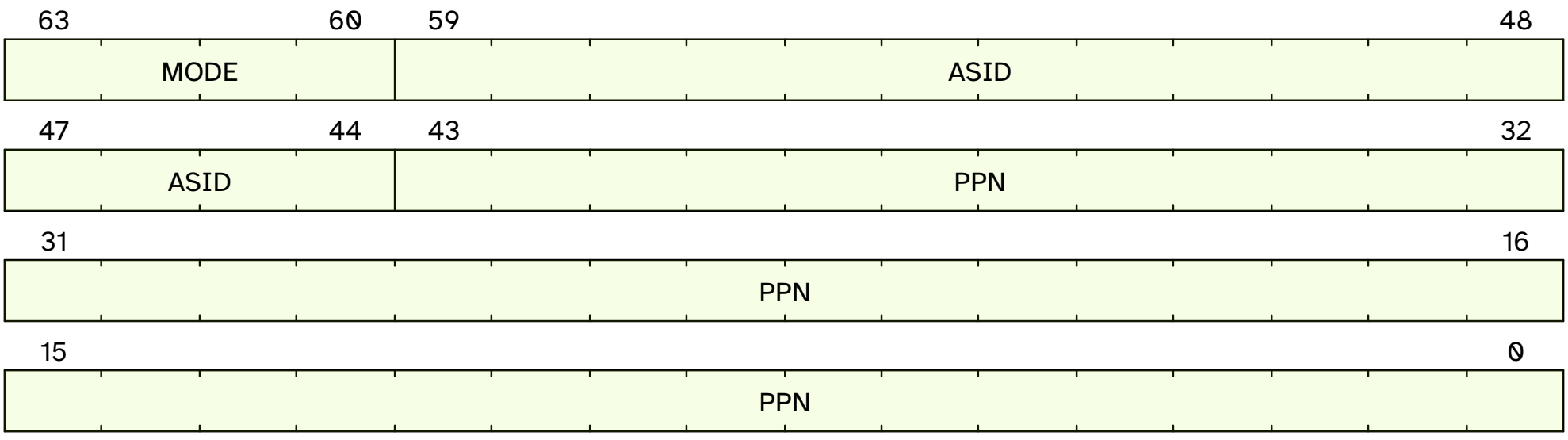


Figure 110. satp Format when CSR[mstatus].SXL == 1

C.109.3. Field Summary

Name	Location	Type	Reset Value
satp.MODE	31 63:60	RW-R	UNDEFINED_LEGAL
satp.ASID	30:22 59:44	RW-R	UNDEFINED_LEGAL
satp.PPN	21:0 43:0	RW-R	UNDEFINED_LEGAL

C.109.4. Fields

satp.MODE Field

Location:

- *31 when CSR[mstatus].SXL == 0*
- *63:60 when CSR[mstatus].SXL == 1*

Description:

Translation Mode

Controls the current translation mode according to the table below.

[separator="!",%autowidth]
!===
! Value ! Name ! Description

! 0 ! Bare a! No translation → virtual address == physical address
<%- if ext?(:Sv39) -%>
! 8 ! Sv39 ! 39-bit virtual address translation
<%- end -%>
<%- if ext?(:Sv48) -%>
! 9 ! Sv48 ! 48-bit virtual address translation
<%- end -%>
<%- if ext?(:Sv57) -%>
! 10 ! Sv57 ! 57-bit virtual address translation
<%- end -%>
!===

Any other value shall be ignored on a write.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

satp.ASID Field

Location:

- *30:22 when CSR[mstatus].SXL == 0*
- *59:44 when CSR[mstatus].SXL == 1*

Description:

Address Space ID

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

satp.PPN Field

Location:

- *21:0 when CSR[mstatus].SXL == 0*
- *43:0 when CSR[mstatus].SXL == 1*

Description:

Physical Page Number

The physical address of the active root page table is PPN << 12.

Can only hold values that correspond to a valid page table base, which will be implementation-dependent.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

C.109.5. 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:

```
MODE = if (SATP_MODE_BARE) {
  if (csr_value.MODE == 0) {
    # In Bare, ASID and PPN must be zero, else the entire write is ignored
    if (csr_value.ASID == 0 && csr_value.PPN == 0) {
```

```
      if (CSR[satp].MODE != 0) {
        # changes *to* Bare mode take effect immediately without needing sfence.vma
        # thus, an implicit sfence.vma occurs now
        VmaOrderType order_type;
        order_type.global = true;
        order_pgtbl_writes_before_vmafence(order_type);

        invalidate_translations(order_type);

        order_pgtbl_reads_after_vmafence(order_type);
      }
      return csr_value.MODE;
    } else {
      return UNDEFINED_LEGAL_DETERMINISTIC;
    }
  }
}
else if (implemented?(ExtensionName::Sv39) && csr_value.MODE == 8) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }
  return csr_value.MODE;
}
else if (implemented?(ExtensionName::Sv48) && csr_value.MODE == 9) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }

  return csr_value.MODE;
}
else if (implemented?(ExtensionName::Sv57) && csr_value.MODE == 10) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }

  return csr_value.MODE;
}
else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}

ASID = if (csr_value.MODE == 0) {
  # when MODE == Bare, PPN and ASID must be zero
  if (csr_value.ASID == 0 && csr_value.PPN == 0) {
    return csr_value.ASID;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else {
  XReg shamt = (xlen() == 32 || (CSR[mstatus].SXL == $bits(XRegWidth::XLEN32))) ? 9 : 16;
  XReg all_ones = ((MXLEN'1 << shamt) - 1);
```

```
  XReg largest_allowed_asid = (MXLEN'1 << shamt) - 1;

  if (csr_value.ASID == all_ones) {
    # the specification states that if all 1's are written to the ASID field, then
    # you must return the largest asid
    return largest_allowed_asid;
  } else if (csr_value.ASID > largest_allowed_asid) {
    # ... but is silent on what happens on any other illegal value
    return UNDEFINED_LEGAL_DETERMINISTIC;
  } else {
    # unrestricted
    return csr_value.ASID;
  }
}

PPN = if (csr_value.MODE == 0) {
  # when MODE == Bare, PPN and ASID must be zero
  if (csr_value.ASID == 0 && csr_value.PPN == 0) {
    return csr_value.PPN;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else {
  # unrestricted
  return csr_value.PPN;
}
```

C.110. scause

Supervisor Cause

Reports the cause of the latest exception.

C.110.1. Attributes

CSR Address	0x142
Defining extension	• S, version >= S@1.11.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.110.2. Format

This CSR format changes dynamically with XLEN.

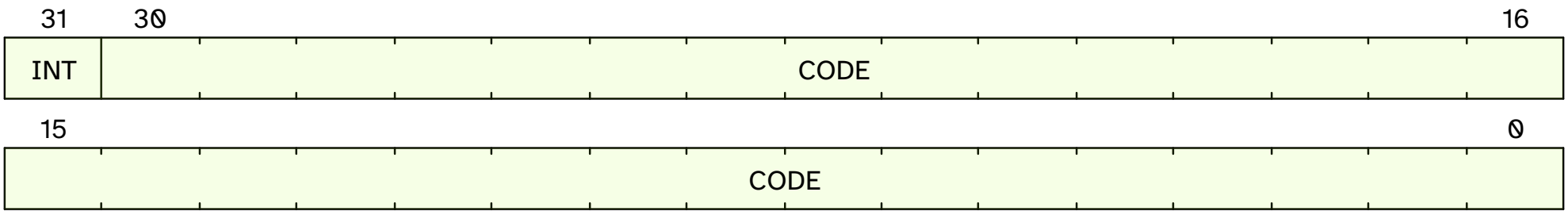


Figure 111. scause Format when CSR[mstatus].SXL == 0

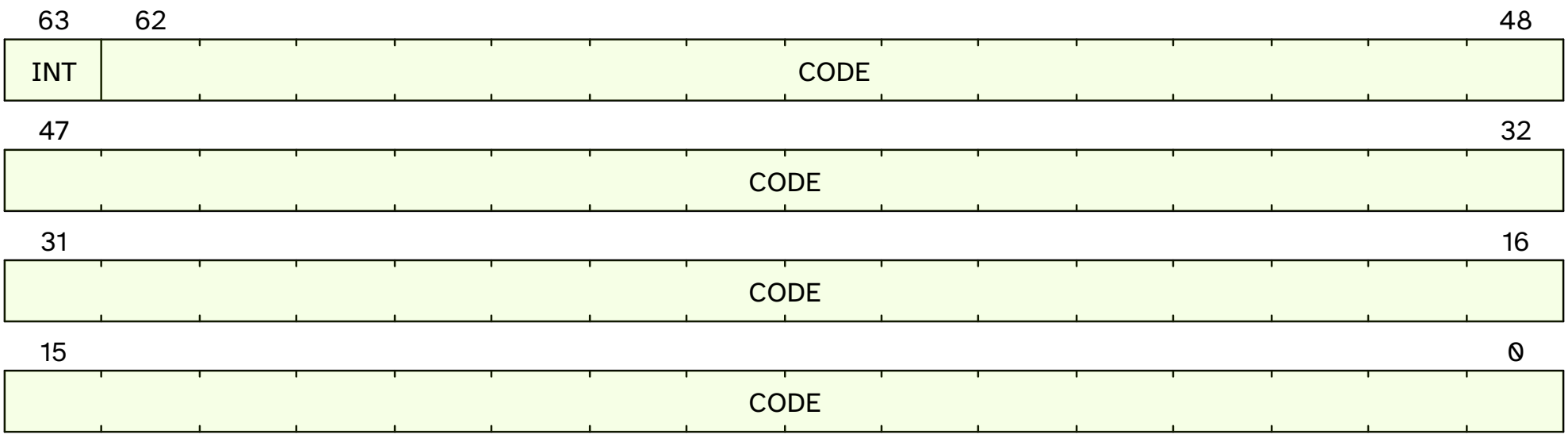


Figure 112. scause Format when CSR[mstatus].SXL == 1

C.110.3. Field Summary

Name	Location	Type	Reset Value
scause.INT	31 63	RW-RH	UNDEFINED_LEGAL
scause.CODE	30:0 62:0	RW-RH	UNDEFINED_LEGAL

C.110.4. Fields

scause.INT Field

Location:

- *31 when CSR[mstatus].SXL == 0*
- *63 when CSR[mstatus].SXL == 1*

Description:

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

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

scause.INT is writable.

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

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

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

scause.CODE Field

Location:

- *30:0 when CSR[mstatus].SXL == 0*
- *62:0 when CSR[mstatus].SXL == 1*

Description:

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

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

scause.CODE is writable.

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

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

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

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

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

C.110.5. 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
# otherwise, the old value is retained
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
# otherwise, the old value is retained
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.111. scounteren

Supervisor Counter Enable

Delegates control of the hardware performance-monitoring counters to U-mode

C.111.1. Attributes

CSR Address	0x106
Defining extension	• S, version >= S@1.11.0
Length	32-bit
Privilege Mode	S

C.111.2. Format

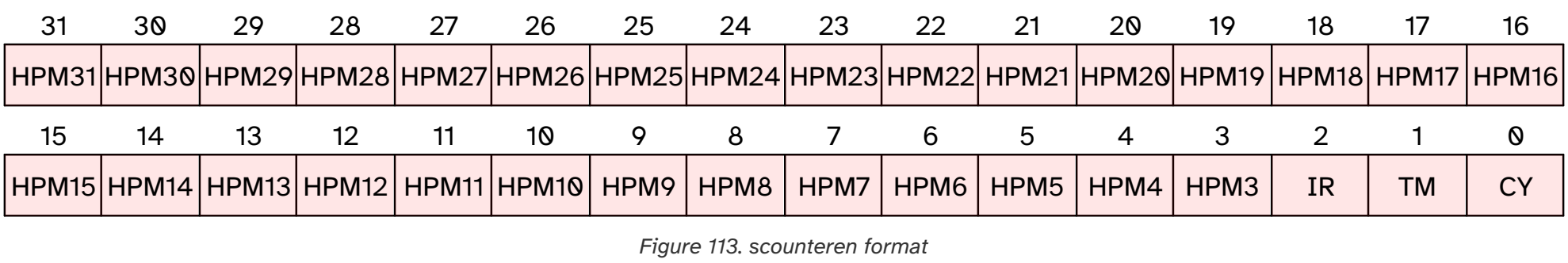


Figure 113. scounteren format

C.111.3. Field Summary

Name	Location	Type	Reset Value
scounteren.CY	0	RW RO	UNDEFINED_LEGAL
scounteren.TM	1	RW RO	UNDEFINED_LEGAL
scounteren.IR	2	RW RO	UNDEFINED_LEGAL
scounteren.HPM3	3	RW RO	UNDEFINED_LEGAL
scounteren.HPM4	4	RW RO	UNDEFINED_LEGAL
scounteren.HPM5	5	RW RO	UNDEFINED_LEGAL
scounteren.HPM6	6	RW RO	UNDEFINED_LEGAL
scounteren.HPM7	7	RW RO	UNDEFINED_LEGAL
scounteren.HPM8	8	RW RO	UNDEFINED_LEGAL
scounteren.HPM9	9	RW RO	UNDEFINED_LEGAL
scounteren.HPM10	10	RW RO	UNDEFINED_LEGAL
scounteren.HPM11	11	RW RO	UNDEFINED_LEGAL
scounteren.HPM12	12	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
scounteren.HPM13	13	RW RO	UNDEFINED_LEGAL
scounteren.HPM14	14	RW RO	UNDEFINED_LEGAL
scounteren.HPM15	15	RW RO	UNDEFINED_LEGAL
scounteren.HPM16	16	RW RO	UNDEFINED_LEGAL
scounteren.HPM17	17	RW RO	UNDEFINED_LEGAL
scounteren.HPM18	18	RW RO	UNDEFINED_LEGAL
scounteren.HPM19	19	RW RO	UNDEFINED_LEGAL
scounteren.HPM20	20	RW RO	UNDEFINED_LEGAL
scounteren.HPM21	21	RW RO	UNDEFINED_LEGAL
scounteren.HPM22	22	RW RO	UNDEFINED_LEGAL
scounteren.HPM23	23	RW RO	UNDEFINED_LEGAL
scounteren.HPM24	24	RW RO	UNDEFINED_LEGAL
scounteren.HPM25	25	RW RO	UNDEFINED_LEGAL
scounteren.HPM26	26	RW RO	UNDEFINED_LEGAL
scounteren.HPM27	27	RW RO	UNDEFINED_LEGAL
scounteren.HPM28	28	RW RO	UNDEFINED_LEGAL
scounteren.HPM29	29	RW RO	UNDEFINED_LEGAL
scounteren.HPM30	30	RW RO	UNDEFINED_LEGAL
scounteren.HPM31	31	RW RO	UNDEFINED_LEGAL

C.111.4. Fields

scounteren.CY Field

Location:

0

Description:

When both scounteren.CY and mcounteren.CY are set, the cycle CSR (an alias of mcycle) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.CY)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.TM Field

Location:

1

Description:

When both scounteren.TM and mcounteren.TM are set, the time CSR (an alias of `mtime` memory-mapped CSR) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.TM)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.IR Field

Location:

2

Description:

When both scounteren.IR and mcounteren.IR are set, the instret CSR (an alias of memory-mapped minstret) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.IR)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM3 Field

Location:

3

Description:

When both scounteren.HPM3 and mcounteren.HPM3 are set, the hpmcounter3 CSR (an alias of mhpmcounter3) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM3)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM4 Field

Location:

4

Description:

When both scounteren.HPM4 and mcounteren.HPM4 are set, the hpmcounter4 CSR (an alias of mhpmcounter4) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM4)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM5 Field

Location:

5

Description:

When both scounteren.HPM5 and mcounteren.HPM5 are set, the hpmcounter5 CSR (an alias of mhpmcounter5) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM5)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM6 Field

Location:

6

Description:

When both scounteren.HPM6 and mcounteren.HPM6 are set, the hpmcounter6 CSR (an alias of mhpmcounter6) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM6)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM7 Field

Location:

7

Description:

When both scounteren.HPM7 and mcounteren.HPM7 are set, the hpmcounter7 CSR (an alias of mhpmcounter7) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM7)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM8 Field

Location:

8

Description:

When both scounteren.HPM8 and mcounteren.HPM8 are set, the hpmcounter8 CSR (an alias of mhpmcounter8) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM8)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM9 Field

Location:

9

Description:

When both scounteren.HPM9 and mcounteren.HPM9 are set, the hpmcounter9 CSR (an alias of mhpmcounter9) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM9)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM10 Field

Location:

10

Description:

When both scounteren.HPM10 and mcounteren.HPM10 are set, the hpmcounter10 CSR (an alias of mhpmcounter10) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM10)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM11 Field

Location:

11

Description:

When both scounteren.HPM11 and mcounteren.HPM11 are set, the hpmcounter11 CSR (an alias of mhpmcounter11) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM11)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM12 Field

Location:

12

Description:

When both scounteren.HPM12 and mcounteren.HPM12 are set, the hpmcounter12 CSR (an alias of mhpmcounter12) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM12)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM13 Field

Location:

13

Description:

When both scounteren.HPM13 and mcounteren.HPM13 are set, the hpmcounter13 CSR (an alias of mhpmcounter13) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM13)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM14 Field

Location:

14

Description:

When both scounteren.HPM14 and mcounteren.HPM14 are set, the hpmcounter14 CSR (an alias of mhpmcounter14) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM14)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM15 Field

Location:

15

Description:

When both scounteren.HPM15 and mcounteren.HPM15 are set, the hpmcounter15 CSR (an alias of mhpmcounter15) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM15)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM16 Field

Location:

16

Description:

When both scounteren.HPM16 and mcounteren.HPM16 are set, the hpmcounter16 CSR (an alias of mhpmcounter16) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM16)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM17 Field

Location:

17

Description:

When both scounteren.HPM17 and mcounteren.HPM17 are set, the hpmcounter17 CSR (an alias of mhpmcounter17) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM17)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM18 Field

Location:

18

Description:

When both scounteren.HPM18 and mcounteren.HPM18 are set, the hpmcounter18 CSR (an alias of mhpmcounter18) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM18)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM19 Field

Location:

19

Description:

When both scounteren.HPM19 and mcounteren.HPM19 are set, the hpmcounter19 CSR (an alias of mhpmcounter19) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM19)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM20 Field

Location:

20

Description:

When both scounteren.HPM20 and mcounteren.HPM20 are set, the hpmcounter20 CSR (an alias of mhpmcounter20) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM20)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM21 Field

Location:

21

Description:

When both scounteren.HPM21 and mcounteren.HPM21 are set, the hpmcounter21 CSR (an alias of mhpmcounter21) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM21)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM22 Field

Location:

22

Description:

When both scounteren.HPM22 and mcounteren.HPM22 are set, the hpmcounter22 CSR (an alias of mhpmcounter22) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM22)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM23 Field

Location:

23

Description:

When both scounteren.HPM23 and mcounteren.HPM23 are set, the hpmcounter23 CSR (an alias of mhpmcounter23) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM23)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM24 Field

Location:

24

Description:

When both scounteren.HPM24 and mcounteren.HPM24 are set, the hpmcounter24 CSR (an alias of mhpmcounter24) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM24)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM25 Field

Location:

25

Description:

When both scounteren.HPM25 and mcounteren.HPM25 are set, the hpmcounter25 CSR (an alias of mhpmcounter25) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM25)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM26 Field

Location:

26

Description:

When both scounteren.HPM26 and mcounteren.HPM26 are set, the hpmcounter26 CSR (an alias of mhpmcounter26) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM26)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM27 Field

Location:

27

Description:

When both scounteren.HPM27 and mcounteren.HPM27 are set, the hpmcounter27 CSR (an alias of mhpmcounter27) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM27)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM28 Field

Location:

28

Description:

When both scounteren.HPM28 and mcounteren.HPM28 are set, the hpmcounter28 CSR (an alias of mhpmcounter28) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM28)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM29 Field

Location:

29

Description:

When both scounteren.HPM29 and mcounteren.HPM29 are set, the hpmcounter29 CSR (an alias of mhpmcounter29) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM29)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM30 Field

Location:

30

Description:

When both scounteren.HPM30 and mcounteren.HPM30 are set, the hpmcounter30 CSR (an alias of mhpmcounter30) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM30)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM31 Field

Location:

31

Description:

When both scounteren.HPM31 and mcounteren.HPM31 are set, the hpmcounter31 CSR (an alias of mhpmcounter31) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM31)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

C.112. senvcfg

Supervisor Environment Configuration

Contains fields that control certain characteristics of the U-mode execution environment.

C.112.1. Attributes

CSR Address	0x10a

Defining extension	• allOf:
	  • S, version >=1.12
	  • U, version >= U@1.0.0

Length	64-bit
Privilege Mode	S

C.112.2. Format

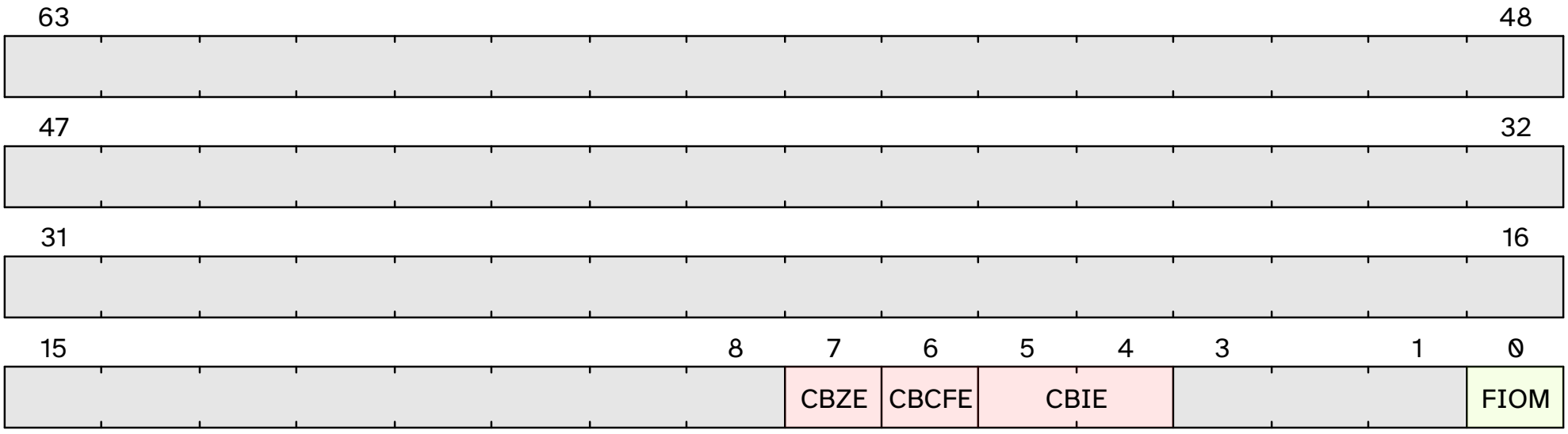


Figure 114. senvcfg format

C.112.3. Field Summary

Name	Location	Type	Reset Value
senvcfg.CBZE	7	RW	UNDEFINED_LEGAL
senvcfg.CBCFE	6	RW	UNDEFINED_LEGAL
senvcfg.CBIE	5:4	RW-R	UNDEFINED_LEGAL
senvcfg.FIOM	0	RW	0

C.112.4. Fields

senvcfg.CBZE Field

Location:

7

Description:

Cache Block Zero instruction Enable

Bit is read-only 0 when menvcfg.CBZE is clear.

*Enables the execution of the cache block zero instruction, cbo.zero,
in U-mode<% if ext?(:H) %> and (in conjunction with henvcfg.CBZE) VU-mode<% end %>.*

- *0: The instruction raises an illegal instruction<% if ext?(:H) %> or virtual instruction exception<% end %>*
- *1: The instruction is executed*

To summarize access:
[separator="!",%autowidth]
!===
! menvcfg.CBZE ! senvcfg.CBZE behavior

! 0 ! read-only 0
! 1
a! writable, independent bit from menvcfg.CBZE
!===

See cbo.zero for a summary of the effect.

Type:

RW

Reset value:

UNDEFINED_LEGAL

senvcfg.CBCFE Field

Location:

6

Description:

Cache Block Clean and Flush instruction Enable

Enables the execution of the cache block clean instruction, cbo.clean, and the
cache block flush instruction, cbo.flush,
<% if ext?(:S) %>
in S-mode
<% elsif ext?(:U) %>
in U-mode
<% end %>.

- `0`: *The instruction raises an illegal instruction <% if ext?(:H) %>or virtual instruction exception<% end %>*
- `1`: *The instruction is executed*

To summarize access:
[separator="!",%autowidth]
!===
! menvcfg.CBCFE ! senvcfg.CBCFE behavior

! 0 ! read-only 0
! 1
a! writable, independent bit from menvcfg.CBCFE
!===

See cbo.clean and/or cbo.flush for a summary of the effect.

Type:

RW

Reset value:

UNDEFINED_LEGAL

senvcfg.CBIE Field

Location:

5:4

Description:

Cache Block Invalidate instruction Enable

This field has restricted values based on the value of menvcfg.CBIE.
When an invalid value is written, it is ignored and the field remains unchanged.

[separator="!",%autowidth,cols=",.>"]
!===
! menvcfg.CBIE ! Valid values of senvcfg.CBIE

! 00 ! 00
! 01 ! 00, 01
! 11 ! 00, 01, 11
!===

Controls execution of the cache block invalidate instruction, cbo.inval,
in U-mode
<% if ext?(:H) %>
and VU-mode (together with henvcfg.CBIE)
<% end %>

1. + +
   - `00`: *The instruction raises an illegal instruction or virtual instruction exception*
   - `01`: *The instruction is executed and performs a flush operation*
   - `10`: *Reserved*
   - `11`: *The instruction is executed and performs an invalidate operation*

   See cbo.inval for more details.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

senvcfg.FIOM Field

Location:

`0`

Description:

Fence of I/O implies Memory

When either senvcfg.FIOM or menvcfg.FIOM is set,
FENCE instructions ordering I/O regions also implicitly order memory regions when executed
in U-mode as follows:

[separator="!",%autowidth,float="center",align="center",cols="^,<",options="header"]
!===
!Instruction bit !Meaning when set
!PI +
PO
!Predecessor device input and memory reads (PR implied) +
Predecessor device output and memory writes (PW implied)
!SI +
SO
!Successor device input and memory reads (SR implied) +
Successor device output and memory writes (SW implied)
!===

Similarly, in U-mode when FIOM=1, if an atomic
instruction that accesses a region ordered as device I/O has its aq
and/or rl bit set, then that instruction is ordered as though it
accesses both device I/O and memory.

See fence for more details.

Type:

RW

Reset value:

`0`

C.112.5. 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:

```
CBZE = csr_value.CBZE
```

```
CBCFE = csr_value.CBCFE
CBIE = if (csr_value.CBIE == 0 || csr_value.CBIE == 1 || csr_value.CBIE == 3) {
  return csr_value.CBIE;
} else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}

FIOM = csr_value.FIOM
```

C.113. sepc

Supervisor Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in (H)S-mode.

Also controls where the hart jumps on an exception return from (H)S-mode.

C.113.1. Attributes

CSR Address	0x141
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

C.113.2. Format

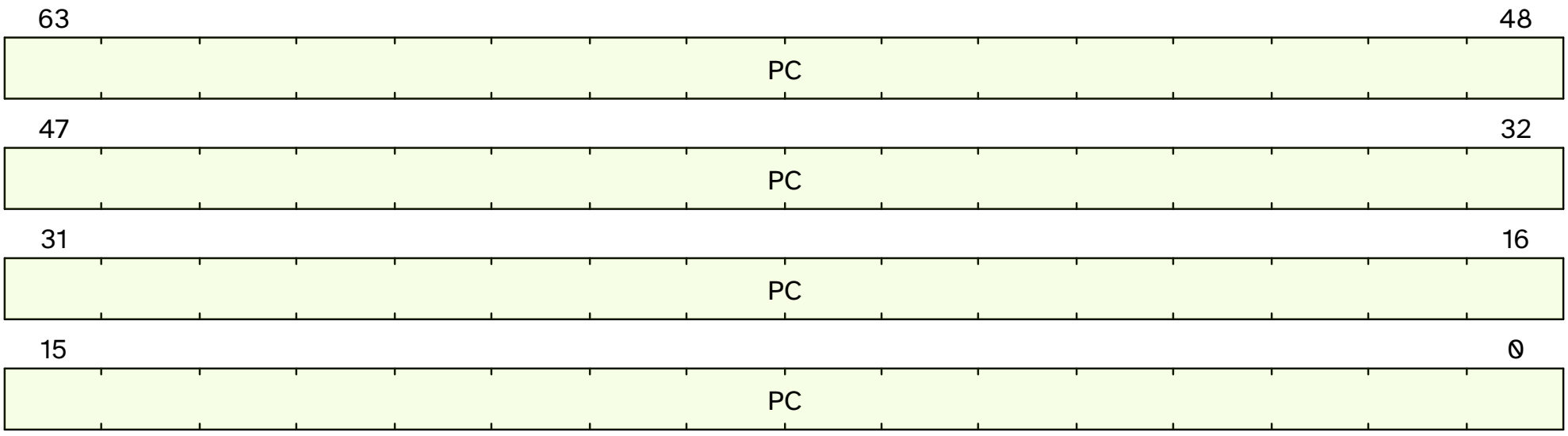


Figure 115. sepc format

C.113.3. Field Summary

Name	Location	Type	Reset Value
sepc.PC	63:0	RW-RH	UNDEFINED_LEGAL

C.113.4. Fields

sepc.PC Field

Location:

63:0

Description:

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

On an exception return from S-mode (from the SRET instruction), control transfers to the virtual address read out of sepc.PC.

Because PCs are always <% if ext?(:C) %>halfword<% else %>word<% end %>-aligned, <% if ext?(:C) %>bit 0<% else %>bits 1:0<% end %> of sepc.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.

Holds bits 63:<%= ext?(:C) ? 2 : 1 %> of the virtual address associated with an exception.

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

C.113.5. 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.113.6. Software read

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

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

C.114. sip

Supervisor Interrupt Pending

A restricted view of the interrupt pending bits in mip.

Hypervisor-related interrupts (VS-mode interrupts and Supervisor Guest interrupts) are not reflected in sip even though those interrupts can be taken in HS-mode. Instead, they are reported through `hip`.

C.114.1. Attributes

CSR Address	0x144
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

C.114.2. Format

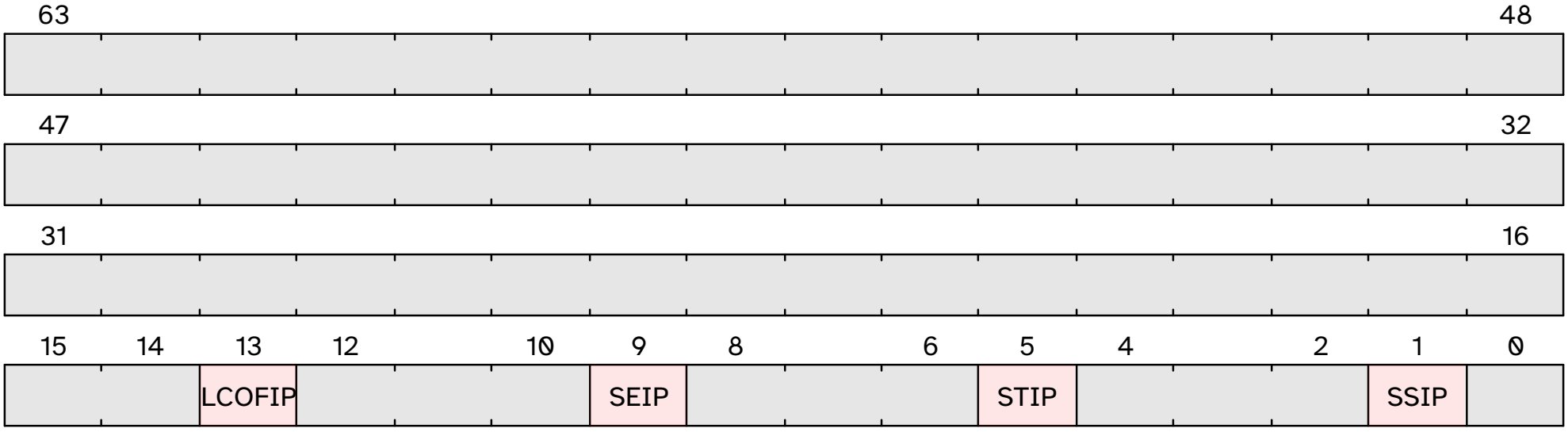


Figure 116. sip format

C.114.3. Field Summary

Name	Location	Type	Reset Value
sip.SSIP	1	RW	UNDEFINED_LEGAL
sip.STIP	5	RO-H	UNDEFINED_LEGAL
sip.SEIP	9	RO-H	UNDEFINED_LEGAL
sip.LCOFIP	13	RW-H	UNDEFINED_LEGAL

C.114.4. Fields

sip.SSIP Field

Location:

1

Description:

Supervisor Software Interrupt Pending

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

When Supervisor Software Interrupts are not delegated to (H)S-mode (mideleg.SSI is clear), sip.SSIP is read-only 0.

Otherwise, sip.SSIP is an alias of mip.SSIP.

<%- 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 S-mode software and, when written, can be used to generate an S-mode Software Interrupt.
<%- end -%>

Since it is an alias, writes to sip.SSIP are also be reflected in mip.SSIP<% if ext?(:Smaia) %> and `mvip.SSIP`*<% end %>.*

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

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

  To summarize:
  [separator="!",%autowidth]
  !===
  ! mideleg.SSI ! sip.SSIP behavior

  ! 0 ! read-only 0
  ! 1 ! writable alias of mip.SSIP <% if ext?(:Smaia) %>and `mvip.SSIP`*<% end %>*
  !===

Type:

RW

Reset value:

UNDEFINED_LEGAL

sip.STIP Field

Location:

5

Description:

Supervisor Timer Interrupt Pending

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

When Supervisor Timer Interrupts are not delegated to (H)S-mode (i.e., mideleg.STI is clear), sip.STIP is read-only 0.

Otherwise, sip.STIP is a read-only view of mip.STIP.

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

- *mip.STIP when mideleg.STI is set*
  <%- if ext?(:Smaia) -%>
- `mvip.STIP` *when mideleg.SSI is set and menvcfg.STCE is clear.*
  <%- end -%>

  To summarize:
  [separator="!",%autowidth]
  !===
  ! mideleg.STI ! sip.STIP behavior

  ! 0 ! read-only 0
  ! 1 ! read-only alias of mip.STIP <% if ext?(:Smaia) %>(and `mvip.STIP` *when menvcfg.STCE is clear)<% end %>*
  !===

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sip.SEIP Field

Location:

9

Description:

Supervisor External Interrupt Pending

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

When Supervisor External Interrupts are not delegated to (H)S-mode (i.e., mideleg.SEI is clear), sip.SEIP is read-only 0.

Otherwise, sip.SEIP is a read-only view of mip.SEIP.

To summarize:
[separator="!",%autowidth]
!===
! mideleg.SEI ! sip.SEIP behavior

! 0 ! read-only 0
! 1 ! read-only alias of mip.SEIP
!===

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sip.LCOFIP Field

Location:

13

Description:

Local Counter Overflow Interrupt pending

Reports the current pending state of a Local Counter Overflow interrupt.

When Local Counter Overflow interrupts are not delegated to (H)S-mode (i.e., mideleg.LCOFI is clear), sip.LCOFIP is read-only 0.

Otherwise, sip.LCOFIP is an alias of mip.LCOFIP.

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

To summarize:
[separator="!",%autowidth]
!===
! mideleg.LCOFI ! sip.LCOFIP behavior

! 0 ! read-only 0
! 1
a! writable alias of mip.LCOFIP (and `vsip.LCOFIP` when `hideleg.LCOFI` is set)
!===

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.115. sscratch

Supervisor Scratch Register

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

C.115.1. Attributes

CSR Address	0x140
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

C.115.2. Format

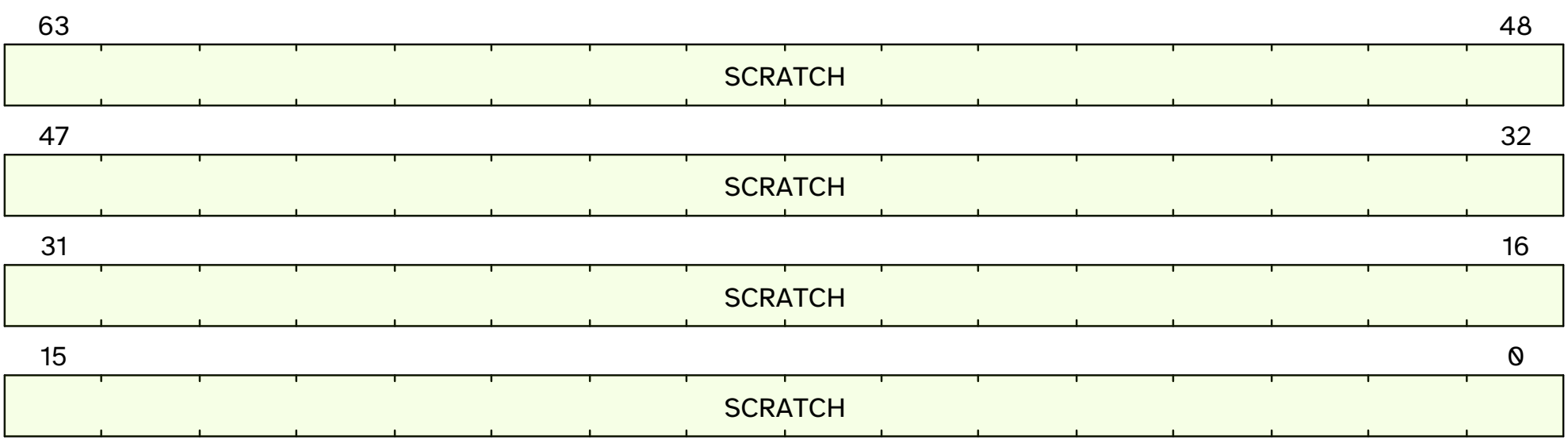


Figure 117. sscratch format

C.115.3. Field Summary

Name	Location	Type	Reset Value
sscratch.SCRATCH	63:0	RW	UNDEFINED_LEGAL

C.115.4. Fields

sscratch.SCRATCH Field

Location:

63:0

Description:

Scratch value

Type:

RW

Reset value:

UNDEFINED_LEGAL

C.116. sstatus

Supervisor Status

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

All fields in sstatus are aliases of the same field in mstatus.

C.116.1. Attributes

CSR Address	0x100
Defining extension	• S, version >= S@1.11.0
Length	32 when CSR[mstatus].SXL == 0 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

C.116.2. Format

This CSR format changes dynamically with XLEN.

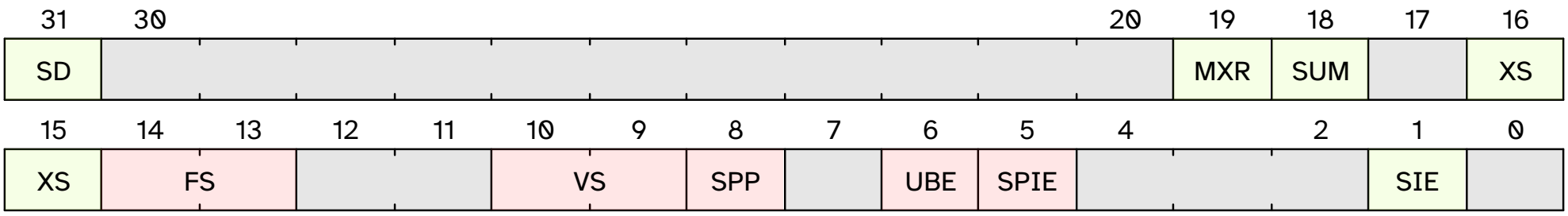


Figure 118. sstatus Format when CSR[mstatus].SXL == 0

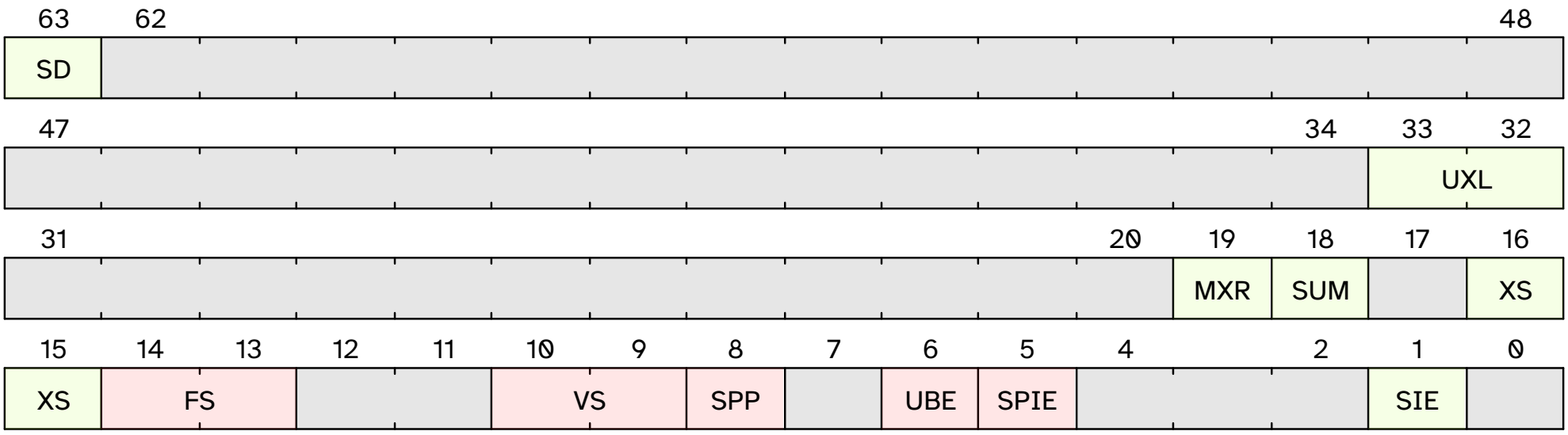


Figure 119. sstatus Format when CSR[mstatus].SXL == 1

C.116.3. Field Summary

Name	Location	Type	Reset Value
sstatus.SD	31 63	RO-H	UNDEFINED_LEGAL
sstatus.UXL	33:32	RO	UNDEFINED_LEGAL
sstatus.MXR	19	RW	UNDEFINED_LEGAL
sstatus.SUM	18	RW	UNDEFINED_LEGAL
sstatus.XS	16:15	RO	UNDEFINED_LEGAL
sstatus.FS	14:13	RW-H	UNDEFINED_LEGAL
sstatus.VS	10:9	RW-H	UNDEFINED_LEGAL
sstatus.SPP	8	RW-H	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
sstatus.UBE	6	RO	UNDEFINED_LEGAL
sstatus.SPIE	5	RW-H	UNDEFINED_LEGAL
sstatus.SIE	1	RW-H	UNDEFINED_LEGAL

C.116.4. Fields

sstatus.SD Field

Location:

- *31 when CSR[mstatus].SXL == 0*
- *63 when CSR[mstatus].SXL == 1*

Description:

State Dirty

Alias of mstatus.SD.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sstatus.UXL Field

UXL is only defined in RV64 (`CSR[mstatus].SXL == 1`)

Location:

33:32

Description:

U-mode XLEN

Alias of mstatus.UXL.

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.MXR Field

Location:

19

Description:

Make eXecutable Readable

Alias of mstatus.MXR.

Type:

RW

Reset value:

UNDEFINED_LEGAL

sstatus.SUM Field

Location:

18

Description:

permit Supervisor Memory Access

Alias of mstatus.SUM.

Type:

RW

Reset value:

UNDEFINED_LEGAL

sstatus.XS Field

Location:

16:15

Description:

Custom (X) extension context Status.

Alias of mstatus.XS.

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.FS Field

Location:

14:13

Description:

Floating point context status.

Alias of mstatus.FS.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.VS Field

Location:

10:9

Description:

Vector context status.

Alias of mstatus.VS.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.SPP Field

Location:

8

Description:

S-mode Previous Privilege

Alias of mstatus.SPP.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.UBE Field

Location:

6

Description:

U-mode Big Endian

Alias of mstatus.UBE.

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.SPIE Field

Location:

5

Description:

S-mode Previous Interrupt Enable

Alias of mstatus.SPIE.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.SIE Field

Location:

1

Description:

S-mode Interrupt Enable

Alias of mstatus.SIE.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

C.117. stval

Supervisor Trap Value

Holds trap-specific information

C.117.1. Attributes

CSR Address	0x143
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

C.117.2. Format

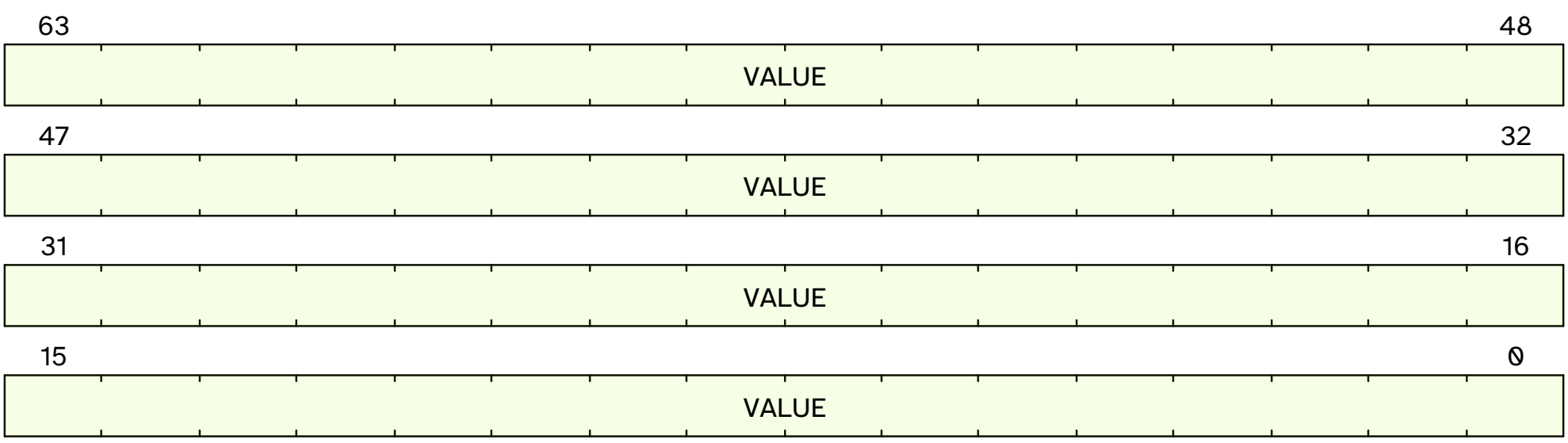


Figure 120. stval format

C.117.3. Field Summary

Name	Location	Type	Reset Value
stval.VALUE	63:0	RW-H	0

C.117.4. Fields

stval.VALUE Field

Location:

63:0

Description:

Written with trap-specific information when a trap is taken into S-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_STVAL_ON_BREAKPOINT == true"]
When caused by an EBREAK instruction, the virtual PC of the breakpoint instruction.

[when,"REPORT_VA_IN_STVAL_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 -%>
! [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 htval.
<%- end -%>
!===

Type:

RW-H

Reset value:

0

C.118. stvec

Supervisor Trap Vector

Controls where traps jump.

C.118.1. Attributes

CSR Address	0x105
Defining extension	• S, version >= S@1.11.0
Length	64-bit
Privilege Mode	S

C.118.2. Format

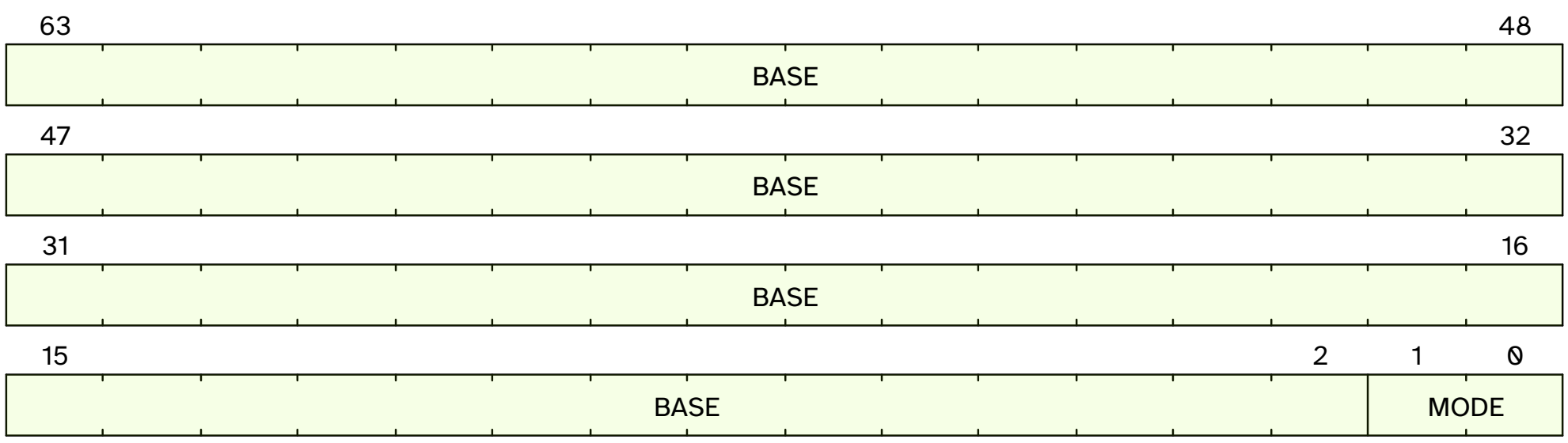


Figure 121. stvec format

C.118.3. Field Summary

Name	Location	Type	Reset Value
stvec.BASE	63:2	RW-R	UNDEFINED_LEGAL
stvec.MODE	1:0	RW-R	0

C.118.4. Fields

stvec.BASE Field

Location:

63:2

Description:

<%- va_size = ext?(:Sv57) ? 57 : (ext?(:Sv48) ? 49 :39) -%>
Bit 63:0 of the virtual address of the exception vector for any trap taken into S-mode.

If the base address is written with a non-cannonical address (i.e., bits 63:<%= va_size %> do not match bit <%= va_size-1 %>), the write should be ignored.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

stvec.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 (stvec.BASE << 2).

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

Type:

RW-R

Reset value:

0

C.118.5. 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
return csr_value.BASE;

MODE = if (STVEC_MODE_DIRECT && csr_value.MODE == 0) {
  return 0;
} else if (STVEC_MODE_VECTORED && csr_value.MODE == 1) {
  return 1;
} else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}
```

C.119. 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.119.1. Attributes

CSR Address	0xc01
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	64-bit
Privilege Mode	U

C.119.2. Format

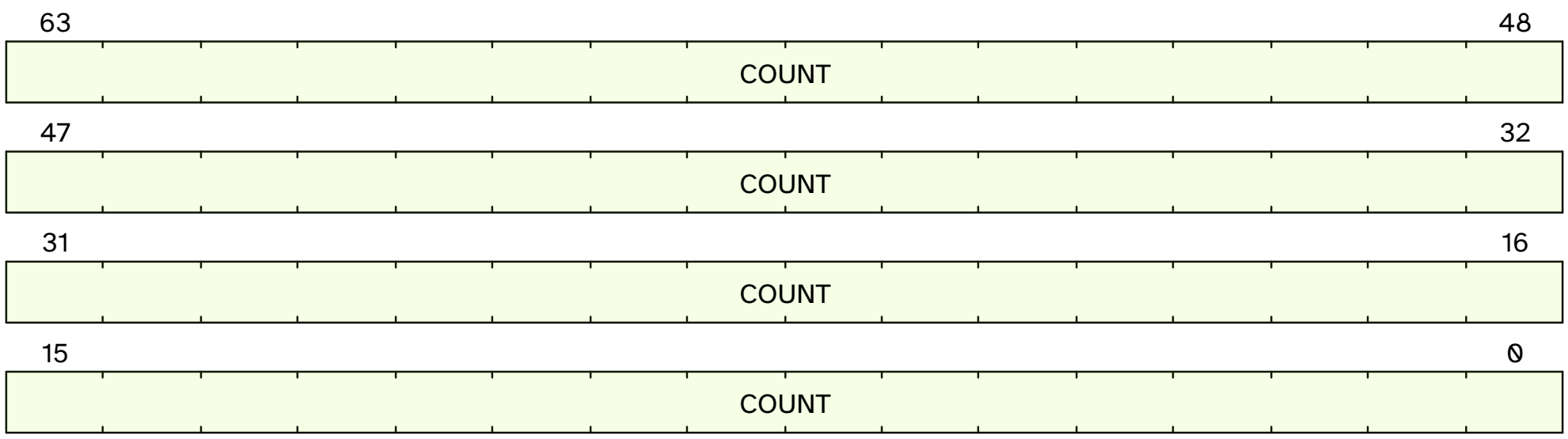


Figure 122. time format

C.119.3. Field Summary

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

C.119.4. 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.119.5. Software read

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

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

C.120. timeh

High-half timer for RDTIME Instruction

timeh is only defined in RV32.

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR `mtimeh`.

Privilege mode access is controlled with mcounteren.TM, scounteren.TM, and hcounteren.TM as follows:

mcounteren.TM	scounteren.TM	scounteren.TM	time behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	Illegal Instruction	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	0	0	read-only	Illegal Instruction	Illegal Instruction	Illegal Instruction
1	1	0	read-only	read-only	Illegal Instruction	Illegal Instruction
1	0	1	read-only	Illegal Instruction	read-only	Illegal Instruction
1	1	1	read-only	read-only	read-only	read-only

C.120.1. Attributes

CSR Address	0xc81
Defining extension	• Zicntr, version >= Zicntr@2.0.0
Length	32-bit
Privilege Mode	U

C.120.2. Format

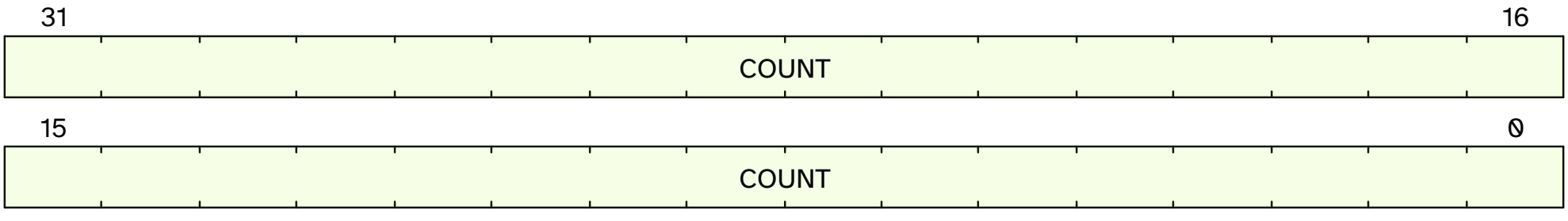


Figure 123. timeh format

C.120.3. Field Summary

Name	Location	Type	Reset Value
timeh.COUNT	31:0	RO-H	UNDEFINED_LEGAL

C.120.4. Fields

timeh.COUNT Field

Location:

31:0

Description:

Reports the most significant 32 bits of the current wall-clock time from the timer device.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

C.120.5. Software read

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

```
if (!TIME_CSR_IMPLEMENTED) {
  unimplemented_csr($encoding);
}
if (mode() == PrivilegeMode::S) {
  if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.TM & scounteren.TM) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.TM == 1'b0 && mcounteren.TM == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.TM & scounteren.TM) == 1'b0) && (mcounteren.IR == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.TM == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mtime()[63:32];
```

Appendix D: IDL Function Details

D.1. implemented? (generated)

Return true if the implementation supports `extension`.

Return Type	`Boolean`
Arguments	`ExtensionName extension`

D.2. implemented_version? (generated)

Return true if the implementation supports `extension` meeting 'version_requirement'.

Return Type	`Boolean`
Arguments	`ExtensionName extension, String version_requirement`

D.3. implemented_csr? (generated)

Return true if csr_addr is an implemented CSR

Return Type	`Boolean`
Arguments	`Bits<12> csr_addr`

D.4. direct_csr_lookup (generated)

Return CSR info for a CSR with direct address csr_addr.

If no CSR exists, <return_value>.valid == false

Return Type	`Csr`
Arguments	`Bits<12> csr_addr`

D.5. indirect_csr_lookup (generated)

Return CSR info for a CSR with indirect address csr_addr at window slot window_slot.

If no CSR exists, <return_value>.valid == false

Return Type	`Csr`
Arguments	`Bits<MXLEN> csr_addr, Bits<4> window_slot`

D.6. csr_hw_read (generated)

Returns the raw value of csr

Return Type	`Bits`
Arguments	`Csr csr`

D.7. csr_sw_read (generated)

Returns the result of CSR[csr].sw_read(); i.e., the software view of the register

Return Type	`Bits`
Arguments	`Csr csr`

D.8. csr_sw_write (generated)

Writes value to csr, applying an WARL transformations first.

Uses the sw_write(...) functions of CSR field definitions.

Return Type	`void`
Arguments	`Csr csr, Bits<MXLEN> value`

D.9. unpredictable (builtin)

Indicate that the hart has reached a state that is unpredictable because the RISC-V spec allows multiple behaviors. Generally, this will be a fatal condition to any emulation, since it is unclear what to do next.

The single argument *why* is a string describing why the hart entered an unpredictable state.

Return Type	`void`
Arguments	`String why`

D.10. read_hpm_counter (builtin)

Returns the value of hpmcounterN.

N must be between 3..31.

hpmcounterN must be implemented.

Return Type	`Bits`
Arguments	`Bits<5> n`

D.11. hartid (builtin)

Returns the value for mhartid as seen by this hart.

Must obey the rules of the priv spec:

> The mhartid CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

Return Type	`XReg`
Arguments	None

D.12. read_mcycle (builtin)

Return the current value of the cycle counter.

Return Type	`Bits`
Arguments	None

D.13. read_mtime (builtin)

Return the current value of the real time device.

Return Type	`Bits`
Arguments	None

D.14. sw_write_mcycle (builtin)

Given a *value* that software is trying to write into mcycle, perform the write and return the value that will actually be written.

Return Type	`Bits`
Arguments	`Bits<64> value`

D.15. cache_block_zero (builtin)

Zero the cache block at the given physical address.

The cache block may be zeroed using 1 or more writes.

A cache-block-sized region is zeroed regardless of whether or not the memory is in a cacheable PMA region.

Return Type	`void`
Arguments	`XReg cache_block_physical_address`

D.16. eei_ecall_from_m (builtin)

When TRAP_ON_ECALL_FROM_M is false, this function will be called to emulate the EEI handling of ECALL-from-M.

If TRAP_ON_ECALL_FROM_M is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.17. eei_ecall_from_s (builtin)

When TRAP_ON_ECALL_FROM_S is false, this function will be called to emulate the EEI handling of ECALL-from-S.

If TRAP_ON_ECALL_FROM_S is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.18. eei_ecall_from_u (builtin)

When TRAP_ON_ECALL_FROM_U is false, this function will be called to emulate the EEI handling of ECALL-from-U.

If TRAP_ON_ECALL_FROM_U is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`

Arguments	None

D.19. eei_ecall_from_vs (builtin)

When TRAP_ON_ECALL_FROM_VS is false, this function will be called to emulate the EEI handling of ECALL-from-VS.

If TRAP_ON_ECALL_FROM_VS is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.20. eei_ebreak (builtin)

When TRAP_ON_EBREAK is false, this function will be called to emulate the EEI handling of EBREAK

If TRAP_ON_EBREAK is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

D.21. memory_model_acquire (builtin)

Perform an acquire; that is, ensure that no subsequent operation in program order appears to an external observer to occur after the operation calling this function.

Return Type	`void`
Arguments	None

D.22. memory_model_release (builtin)

Perform a release; that is, ensure that no prior store in program order can be observed external to this hart after this function returns.

Return Type	`void`
Arguments	None

D.23. assert (builtin)

Assert that a condition is true. Failure represents an error in the IDL model.

Return Type	`void`
Arguments	`Boolean test, String message`

D.24. notify_mode_change (builtin)

Called whenever the privilege mode changes. Downstream tools can use this to hook events.

Return Type	`void`
Arguments	`PrivilegeMode new_mode, PrivilegeMode old_mode`

D.25. abort_current_instruction (builtin)

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

Return Type	`void`
Arguments	None

D.26. ebreak (builtin)

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

Return Type	`void`
Arguments	None

D.27. prefetch_instruction (builtin)

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

Return Type	`void`
Arguments	`XReg virtual_address`

D.28. prefetch_read (builtin)

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

Return Type	`void`
Arguments	`XReg virtual_address`

D.29. prefetch_write (builtin)

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

Return Type	`void`
Arguments	`XReg virtual_address`

D.30. fence (builtin)

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

Return Type	`void`
Arguments	`Boolean pi, Boolean pr, Boolean po, Boolean pw, Boolean si, Boolean sr, Boolean so, Boolean sw`

D.31. fence_tso (builtin)

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

Return Type	`void`
Arguments	None

D.32. ifence (builtin)

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

Return Type	`void`
Arguments	None

D.33. pause (builtin)

Pause hart retirement for a implementation-defined period of time, which may be zero.

See Zihintpause for more.

Return Type	`void`
Arguments	None

D.34. pow (generated)

Return `value` to the power `exponent`.

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

D.35. maybe_cache_translation (generated)

Given a translation result, potentially cache the result for later use. This function models a TLB fill operation. A valid implementation does nothing.

Return Type	`void`
Arguments	`XReg vaddr, MemoryOperation op, TranslationResult result`

D.36. cached_translation (generated)

Possibly returns a cached translation result matching vaddr.

CachedTranslationResult contains a Boolean 'valid' field. If valid, 'result' is a usable translation. Otherwise, the cache lookup failed.

Return Type	`CachedTranslationResult`
Arguments	`XReg vaddr, MemoryOperation op`

D.37. order_pgtbl_writes_before_vmafence (builtin)

Orders all writes prior to this call in global memory order that affect a page table in the set identified by order_type before any subsequent sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.W.INVAL.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType order_type`

D.38. order_pgtbl_reads_after_vmafence (builtin)

Orders all reads after to this call in global memory order to a page table in the set identified by order_type after any prior sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.INVAL.IR.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType order_type`

D.39. invalidate_translations (generated)

Locally invalidate the cached S-mode/VS-mode/G-stage address translations contained in the set identified by inval_type.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType inval_type`

D.40. read_physical_memory

Read from physical memory.

Return Type	`Bits<len>`
Arguments	`XReg paddr`

```
if (len == 8) {
  return read_physical_memory_8(paddr);
} else if (len == 16) {
  return read_physical_memory_16(paddr);
} else if (len == 32) {
  return read_physical_memory_32(paddr);
} else if (len == 64) {
  return read_physical_memory_64(paddr);
} else {
  assert(false, "Invalid len");
}
```

D.41. read_physical_memory_8 (builtin)

Read a byte from physical memory.

Return Type	`Bits⑧`
Arguments	`XReg paddr`

D.42. read_physical_memory_16 (builtin)

Read two bytes from physical memory.

Return Type	`Bits⑯`

Arguments	`XReg paddr`

D.43. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

D.44. read_physical_memory_64 (builtin)

Read eight bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

D.45. write_physical_memory

Write to physical memory.

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

```
if (len == 8) {
  write_physical_memory_8(paddr, value);
} else if (len == 16) {
  write_physical_memory_16(paddr, value);
} else if (len == 32) {
  write_physical_memory_32(paddr, value);
} else if (len == 64) {
  write_physical_memory_64(paddr, value);
} else {
  assert(false, "Invalid len");
}
```

D.46. write_physical_memory_8 (builtin)

Write a byte to physical memory.

Return Type	`void`
Arguments	`XReg paddr, Bits<8> value`

D.47. write_physical_memory_16 (builtin)

Write two bytes to physical memory.

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

D.48. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

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

D.49. write_physical_memory_64 (builtin)

Write eight bytes to physical memory.

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

D.50. wfi (builtin)

Wait-for-interrupt: hint that the processor should enter a low power state until the next interrupt.

A valid implementation is a no-op.

The model will advance the PC; this function does not need to.

Return Type	`void`
Arguments	None

D.51. pma_applies? (builtin)

Checks if *attr* is applied to the entire physical address region between [paddr, paddr + len) based on static PMA attributes.

Return Type	`Boolean`
Arguments	`PmaAttribute attr, Bits<PHYS_ADDR_WIDTH> paddr, U32 len`

D.52. atomic_check_then_write_32 (builtin)

Atomically:

- Reads 32-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 32-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, Bits<32> compare_value, Bits<32> write_value`

D.53. atomic_check_then_write_64 (builtin)

Atomically:

- Reads 64-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, Bits<64> compare_value, Bits<64> write_value`

D.54. atomically_set_pte_a (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' bit and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len`

D.55. atomically_set_pte_a_d (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' and 'D' bits and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len`

D.56. atomic_read_modify_write_32 (builtin)

Atomically read-modify-write 32-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	`Bits`

Arguments	`Bits<PHYS_ADDR_WIDTH>  phys_addr, Bits<32>  value, AmoOperation op`

D.57. atomic_read_modify_write_64 (builtin)

Atomically read-modify-write 64-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	`Bits`
Arguments	`Bits<PHYS_ADDR_WIDTH>  phys_addr, Bits<64>  value, AmoOperation op`

D.58. set_external_interrupt

Set an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

D.59. clear_external_interrupt

Clear an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

D.60. set_software_interrupt

Set a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

D.61. clear_software_interrupt

Clear a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

D.62. set_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

D.63. clear_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

D.64. refresh_pending_interrupts

refreshes the calculation of a pending interrupt

needs to be called after any state update that could change a pending interrupt. This includes: - CSR[mip] - CSR[mie] - CSR[mstatus].MIE - CSR[mstatus].SIE - CSR[vsstatus].SIE - CSR[mideleg] - CSR[sideleg] - CSR[hideleg] - CSR[hvip] - CSR[hgeip] - CSR[hgeie] - mode changes

Return Type	`void`
Arguments	None

```
Bits<MXLEN> pending_ints = mip.sw_read() & $bits(mie);
if (pending_ints == 0) {
  pending_and_enabled_interrupts = 0;
  return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, ">
1.9.1") && implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_enabled_ints = mode() == PrivilegeMode::M) && (mstatus.MIE == 1'b0 ? 0 : ($bits(mie) & (HAS_MIDELEG ?
~$bits(mideleg) : ~MXLEN'0));
Bits<MXLEN> mmode_pending_and_enabled = pending_ints & mmode_enabled_ints;
if (mmode_pending_and_enabled != 0) {
  pending_and_enabled_interrupts = mmode_pending_and_enabled;
  return ;
}
if (misa.S == 1'b1) {
  Bits<MXLEN> smode_enabled_ints = mode() == PrivilegeMode::M) || (mstatus.SIE == 1'b0 ? 0 : $bits(mie) &
($bits(mideleg));
  Bits<MXLEN> smode_pending_and_enabled = pending_ints & smode_enabled_ints;
  if (smode_pending_and_enabled != 0) {
    pending_and_enabled_interrupts = smode_pending_and_enabled;
    return ;
  }
}
pending_and_enabled_interrupts = 0;
```

D.65. highest_priority_interrupt

Given a bitmask of interrupts in the format of MIE/MIP, return the highest priority interrupt code that is set

Interrupt priority is: MEI, MSI, MTI, SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI

Return Type	`InterruptCode`
Arguments	`Bits<MXLEN> int_mask`

```
if (int_mask[$bits(InterruptCode::MachineExternal)] == 1'b1) {
  return InterruptCode::MachineExternal;
} else if (int_mask[$bits(InterruptCode::MachineSoftware)] == 1'b1) {
  return InterruptCode::MachineSoftware;
} else if (int_mask[$bits(InterruptCode::MachineTimer)] == 1'b1) {
  return InterruptCode::MachineTimer;
}
if (misa.S == 1'b1) {
  if (int_mask[$bits(InterruptCode::SupervisorExternal)] == 1'b1) {
```

```
    return InterruptCode::SupervisorExternal;
  } else if (int_mask[$bits(InterruptCode::SupervisorSoftware)] == 1'b1) {
    return InterruptCode::SupervisorSoftware;
  } else if (int_mask[$bits(InterruptCode::SupervisorTimer)] == 1'b1) {
    return InterruptCode::SupervisorTimer;
  }
}
if (implemented?(ExtensionName::Sscofpmf)) {
  if (int_mask[$bits(InterruptCode::LocalCounterOverflow)] == 1'b1) {
    return InterruptCode::LocalCounterOverflow;
  }
}
assert(false, "There is no valid interrupt");
```

D.66. choose_interrupt

Return the highest priority interrupt that is both pending and enabled and the mode it will be taken in

Return Type	`InterruptCode, PrivilegeMode`
Arguments	None

```
InterruptCode chosen;
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, ">
1.9.1") && implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_pending_and_enabled = pending_and_enabled_interrupts & ~(HAS_MIDELEG ? $bits(mideleg) : MXLEN'0);
if (mmode_pending_and_enabled != 0) {
  assert((mode() != PrivilegeMode::M) || (mstatus.MIE == 1'b1), "M-mode interrupts are not enabled");
  chosen = highest_priority_interrupt(mmode_pending_and_enabled);
} else if (misa.S == 1'b1) {
  Bits<MXLEN> smode_pending_and_enabled = (pending_and_enabled_interrupts & $bits(mideleg));
  if (smode_pending_and_enabled != 0) {
    assert((mode() == PrivilegeMode::U) || (mode() == PrivilegeMode::VU) || (mode() == PrivilegeMode::VS) || (mode() ==
PrivilegeMode::S) && (mstatus.SIE == 1'b1), "S-mode interrupt can't be triggered");
    chosen = highest_priority_interrupt(smode_pending_and_enabled);
  }
}
assert($bits(chosen) != 0, "Didn't pick interrupt?");
PrivilegeMode to_mode;
Bits<MXLEN> chosen_mask = (MXLEN'1 << $bits(chosen));
if (((HAS_MIDELEG ? $bits(mideleg) : MXLEN'0) & chosen_mask) == 0) {
  to_mode = PrivilegeMode::M;
} else {
  if (misa.S == 1'b1) {
    to_mode = PrivilegeMode::S;
  } else {
    to_mode = PrivilegeMode::U;
  }
}
return chosen, to_mode;
```

D.67. take_interrupt

Take (adjust CSRs and set PC to handler) the highest priority interrupt that is both pending and enabled

Return Type	`void`
Arguments	None

```
PrivilegeMode to_mode;
InterruptCode code;
(code, to_mode = choose_interrupt());
if (to_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
  mstatus.MPP = $bits(mode())[1:0];
  if (misa.H == 1'b1) {
    mstatus.MPV = $bits(mode())[2];
    mtval2.VALUE = 0;
```

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

D.68. fetch_memory_aligned_16

Fetch 16 bits from virtual memory using a known aligned address.

Return Type	`Bits⑯`
Arguments	`XReg virtual_address`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 16, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<16>(result.paddr);
```

D.69. fetch_memory_aligned_32

Fetch 32 bits from virtual memory using a known aligned address.

Return Type	`Bits`
Arguments	`XReg virtual_address`

```
TranslationResult result;
```

```
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 32, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<32>(result.paddr);
```

D.70. power_of_2?

Returns true if value is a power of two, false otherwise

Return Type	`Boolean`
Arguments	`Bits<N> value`

```
return (value != 0) && value & (value - 1 == 0);
```

D.71. ary_includes?

Returns true if *value* is an element of ary, and false otherwise

Return Type	`Boolean`
Arguments	`Bits<ELEMENT_SIZE> ary[ARY_SIZE], Bits<ELEMENT_SIZE> value`

```
for (U32 i = 0; i < ARY_SIZE; i++) {
  if (ary[i] == value) {
    return true;
  }
}
return false;
```

D.72. has_virt_mem?

Returns true if some virtual memory translation (Sv*) is supported in the config.

Return Type	`Boolean`
Arguments	None

```
return implemented?(ExtensionName::Sv32) || implemented?(ExtensionName::Sv39) || implemented?(ExtensionName::Sv48) ||
implemented?(ExtensionName::Sv57);
```

D.73. highest_set_bit

Returns the position of the highest (nearest MSB) bit that is '1', or -1 if value is zero.

Return Type	`XReg`
Arguments	`XReg value`

```
for (U32 i = xlen() - 1; i >= 0; i--) {
  if (value[i] == 1) {
    return i;
  }
```

```
}
return -'sd1;
```

D.74. lowest_set_bit

Returns the position of the lowest (nearest LSB) bit that is '1', or XLEN if value is zero.

Return Type	`XReg`
Arguments	`XReg value`

```
for (U32 i = 0; i < xlen(); i++) {
  if (value[i] == 1) {
    return i;
  }
}
return xlen();
```

D.75. bit_length

Returns the minimum number of bits needed to represent value.

Only works on unsigned values.

The value `0` returns 1.

Return Type	`XReg`
Arguments	`XReg value`

```
for (XReg i = 63; i > 0; i--) {
  if (value[i] == 1) {
    return i;
  }
}
return 1;
```

D.76. count_leading_zeros

Returns the number of leading `0` bits before the most-significant 1 bit of value, or N if value is zero.

Return Type	`Bits<bit_length(N)>`
Arguments	`Bits<N> value`

```
for (U32 i = 0; i < N; i++) {
  if (value[N - 1 - i] == 1) {
    return i;
  }
}
return N;
```

D.77. sext

Sign extend `value` starting at `first_extended_bit`.

Bits [`XLEN-1:`first_extended_bit`] of the return value should get the value of bit (`first_extended bit - 1`).

Return Type	`XReg`
Arguments	`XReg value, XReg first_extended_bit`

```
if (first_extended_bit == MXLEN) {
  return value;
} else {
  Bits<1> sign = value[first_extended_bit - 1];
  for (U32 i = MXLEN - 1; i >= first_extended_bit; i--) {
    value[i] = sign;
  }
  return value;
}
```

D.78. is_naturally_aligned

Checks if value is naturally aligned to N bits.

Return Type	`Boolean`
Arguments	`XReg value`

```
return true if (N == 8);
XReg Mask = (N / 8) - 1;
return (value & ~Mask) == value;
```

D.79. in_naturally_aligned_region?

Checks if a length-bit access starting at address lies entirely within an N-bit naturally-aligned region.

Return Type	`Boolean`
Arguments	`XReg address, U32  length`

```
XReg Mask = (N / 8) - 1;
return (address & ~Mask) == ((address + length - 1) & ~Mask);
```

D.80. contains?

Given a *region* defined by region_start, region_size, determine if a *target* defined by target_start, target_size is completely contained with the region.

Return Type	`Boolean`
Arguments	`XReg region_start, U32  region_size, XReg target_start, U32  target_size`

```
return target_start >= region_start && (target_start + target_size) <= (region_start + region_size);
```

D.81. set_fp_flag

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

Return Type	`void`

Arguments	`FpFlag flag`

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

D.82. rm_to_mode

Convert rm to a RoundingMode.

encoding is the full encoding of the instruction rm comes from.

Will raise an IllegalInstruction exception if rm is a reserved encoding.

Return Type	`RoundingMode`
Arguments	`Bits<3> rm, Bits<32> encoding`

```
if (rm == $bits(RoundingMode::RNE)) {
  return RoundingMode::RNE;
} else if (rm == $bits(RoundingMode::RTZ)) {
  return RoundingMode::RTZ;
} else if (rm == $bits(RoundingMode::RDN)) {
  return RoundingMode::RDN;
} else if (rm == $bits(RoundingMode::RUP)) {
  return RoundingMode::RUP;
} else if (rm == $bits(RoundingMode::RMM)) {
  return RoundingMode::RMM;
} else if (rm == $bits(RoundingMode::DYN)) {
  return $enum(RoundingMode, fcsr.FRM);
} else {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

D.83. mark_f_state_dirty

Potentially updates mstatus.FS to the Dirty (3) state, depending on configuration settings.

Return Type	`void`
Arguments	None

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

D.84. nan_box

Produces a properly NaN-boxed floating-point value from a floating-point value of smaller size by adding all 1's to the upper bits.

Return Type	`Bits<TO_SIZE>`
Arguments	`Bits<FROM_SIZE> from_value`

```
assert(FROM_SIZE < TO_SIZE, "Bad template arguments; FROM_SIZE must be less than TO_SIZE");
return {{TO_SIZE - FROM_SIZE{1'b1}}, from_value};
```

D.85. check_f_ok

Checks if instructions from the F extension can be executed, and, if not, raise an exception.

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

```
if (MUTABLE_MISA_F && misa.F == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
if (mstatus.FS == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

D.86. is_sp_neg_inf?

Return true if sp_value is negative infinity.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_NEG_INF;
```

D.87. is_sp_pos_inf?

Return true if sp_value is positive infinity.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_POS_INF;
```

D.88. is_sp_neg_norm?

Returns true if sp_value is a negative normal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 1) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

D.89. is_sp_pos_norm?

Returns true if sp_value is a positive normal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 0) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

D.90. is_sp_neg_subnorm?

Returns true if sp_value is a negative subnormal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 1) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

D.91. is_sp_pos_subnorm?

Returns true if sp_value is a positive subnormal number.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[31] == 0) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

D.92. is_sp_neg_zero?

Returns true if sp_value is negative zero.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_NEG_ZERO;
```

D.93. is_sp_pos_zero?

Returns true if sp_value is positive zero.

Return Type	`Boolean`

Arguments	`Bits<32> sp_value`

```
return sp_value == SP_POS_ZERO;
```

D.94. is_sp_nan?

Returns true if sp_value is a NaN (quiet or signaling)

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22:0] != 0);
```

D.95. is_sp_signaling_nan?

Returns true if sp_value is a signaling NaN

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 0) && (sp_value[21:0] != 0);
```

D.96. is_sp_quiet_nan?

Returns true if sp_value is a quiet NaN

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 1);
```

D.97. softfloat_shiftRightJam32

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned. The value of dist can be arbitrarily large. In particular, if dist is greater than 32, the result will be either 0 or 1, depending on whether a is zero or nonzero.

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

```
return (dist < 31) ? a >> dist | (a << (-dist & 31 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

D.98. softfloat_shiftRightJam64

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned.

The value of 'dist' can be arbitrarily large. In particular, if dist is greater than 64, the result will be either 0 or 1, depending on whether a is zero or nonzero.

Return Type	`Bits`
Arguments	`Bits<64> a, Bits<32> dist`

```
return (dist < 63) ? a >> dist | (a << (-dist & 63 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

D.99. softfloat_roundToI32

Round to signed 32-bit integer, using rounding_mode

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
    roundIncrement = 0xFFF;
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 0) != (sign == 1) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

D.100. softfloat_roundToUI32

Round to unsigned 32-bit integer, using rounding_mode

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1) {
    if (sig == 0) {
      return 0;
    }
    if (roundingMode == RoundingMode::RDN) {
      set_fp_flag(FpFlag::NV);
    }
```

```
  } else {
    if (roundingMode == RoundingMode::RUP) {
      roundIncrement = 0xFFF;
    }
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? UI32_NEG_OVERFLOW : UI32_POS_OVERFLOW;
}
Bits<32> z = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  z = z & ~32'b1;
}
if ((z != 0) && (sign == 1)) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? UI32_NEG_OVERFLOW : UI32_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

D.101. packToF32UI

Pack components into a 32-bit value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig`

```
return {sign, exp, sig};
```

D.102. packToF16UI

Pack components into a 16-bit value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<5> exp, Bits<10> sig`

```
return {sign, exp, sig};
```

D.103. softfloat_normSubnormalF16Sig

normalize subnormal half-precision value

Return Type	`Bits<5>, Bits⑩`
Arguments	`Bits<16> hp_value`

```
Bits<8> shift_dist = count_leading_zeros<16>(hp_value);
return 1 - shift_dist, hp_value << shift_dist;
```

D.104. softfloat_roundPackToF32

Round FP value according to mdode and then pack it in IEEE format.

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
  roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
  if ($signed(exp) < 0) {
    Boolean isTiny = ($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 (0xFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
    set_fp_flag(FpFlag::OF);
    set_fp_flag(FpFlag::NX);
    return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0) ? 1 : 0);   } } sig = (sig + roundIncrement);
if (sig == 0) {
  exp = 0;
}
return packToF32UI(sign, exp, sig);
```

D.105. softfloat_normRoundPackToF32

Normalize, round, and pack into a 32-bit floating point value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
  return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
  return softfloat_roundPackToF32(sign, exp, sig << shiftDist, mode);
}
```

D.106. signF32UI

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

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

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

D.107. expF32UI

Extract exponent of a 32-bit floating point number

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

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

D.108. fracF32UI

Extract significand of a 32-bit floating point number

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

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

D.109. returnNonSignalingNaN

Returns a non-signalling NaN version of the floating-point number Does not modify the input

Return Type	`U32`
Arguments	`U32 a`

```
U32 a_copy = a;
a_copy[22] = 1'b1;
return a_copy;
```

D.110. returnMag

Returns magnitude of the given number Does not modify the input

Return Type	`U32`
Arguments	`U32 a`

```
U32 a_copy = a;
a_copy[31] = 1'b0;
return a_copy;
```

D.111. returnLargerMag

Returns the larger number between a and b by magnitude If either number is signaling NaN then that is made quiet

Return Type	`U32`
Arguments	`U32 a, U32 b`

```
U32 mag_a = returnMag(a);
U32 mag_b = returnMag(b);
```

```
U32 nonsig_a = returnNonSignalingNaN(a);
U32 nonsig_b = returnNonSignalingNaN(b);
if (mag_a < mag_b) {
  return nonsig_b;
}
if (mag_b < mag_a) {
  return nonsig_a;
}
return (nonsig_a < nonsig_b) ? nonsig_a : nonsig_b;
```

D.112. softfloat_propagateNaNF32UI

Interpreting 'a' and 'b' as the bit patterns of two 32-bit floating- | point values, at least one of which is a NaN, returns the bit pattern of | the combined NaN result. If either 'a' or 'b' has the pattern of a | signaling NaN, the invalid exception is raised.

Return Type	`U32`
Arguments	`U32 a, U32 b`

```
Boolean isSigNaN_a = is_sp_signaling_nan?(a);
Boolean isSigNaN_b = is_sp_signaling_nan?(b);
if (isSigNaN_a || isSigNaN_b) {
  set_fp_flag(FpFlag::NV);
}
return SP_CANONICAL_NAN;
```

D.113. softfloat_addMagsF32

Returns sum of the magnitudes of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'd0) {
    z = a + b;
    return z;
  }
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  signZ = signF32UI(a);
  expZ = expA;
  sigZ = 32'h01000000 + sigA + sigB;
  if (sigZ & 0x1) == 0) && (expZ < 8'hFE {
    sigZ = sigZ >> 1;
    return (32'h0 + (signZ << 31) + (expZ << 23) + sigZ);
  }
  sigZ = sigZ << 6;
} else {
  signZ = signF32UI(a);
```

```
U32 sigA_32 = 32'h0 + (sigA << 6);
U32 sigB_32 = 32'h0 + (sigA << 6);
if (expDiff < 0) {
  if (expB == 8'hFF) {
    if (sigB != 0) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return packToF32UI(signZ, 8'hFF, 23'h0);
  }
  expZ = expB;
  sigA_32 = (expA == 0) ? 2 * sigA_32 : (sigA_32 + 0x20000000);
  sigA_32 = softfloat_shiftRightJam32(sigA_32, (32'h0 - expDiff));
} else {
  if (expA == 8'hFF) {
    if (sigA != 0) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  expZ = expA;
  sigB_32 = (expB == 0) ? 2 * sigB_32 : (sigB_32 + 0x20000000);
  sigB_32 = softfloat_shiftRightJam32(sigB_32, (32'h0 + expDiff));
}
U32 sigZ = 0x20000000 + sigA + sigB;
if (sigZ < 0x40000000) {
    expZ = expZ - 1;
    sigZ = sigZ << 1;
  }
}
return softfloat_roundPackToF32(signZ, expZ, sigZ[22:0], mode);
```

D.114. softfloat_subMagsF32

Returns difference of the magnitudes of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
U32 sigDiff;
U32 sigX;
U32 sigY;
U32 sigA_32;
U32 sigB_32;
Bits<8> shiftDist;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  sigDiff = sigA - sigB;
  if (sigDiff == 0) {
    return packToF32UI(((mode == RoundingMode::RDN) ? 1 : 0), 0, 0);
  }
  if (expA != 0) {
    expA = expA - 1;
  }
  signZ = signF32UI(a);
  if (sigDiff < 0) {
```

```
    signZ = ~signZ;
    sigDiff = -32'sh1 * sigDiff;
  }
  shiftDist = count_leading_zeros<32>(sigDiff) - 8;
  expZ = expA - shiftDist;
  if (expZ < 0) {
    shiftDist = expA;
    expZ = 0;
  }
  return packToF32UI(signZ, expZ, sigDiff << shiftDist);
} else {
  signZ = signF32UI(a);
  sigA_32 = 32'h0 + (sigA << 7);
  sigB_32 = 32'h0 + (sigB << 7);
  if (expDiff < 0) {
    signZ = ~signZ;
    if (expB == 0xFF) {
      if (sigB_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, expB, 0);
    }
    expZ = expB - 1;
    sigX = sigB_32 | 0x40000000;
    sigY = sigA_32 + ((expA != 0) ? 0x40000000 : sigA_32);
    expDiff = -expDiff;
  } else {
    if (expA == 0xFF) {
      if (sigA_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
    }
    expZ = expA - 1;
    sigX = sigA_32 | 0x40000000;
    sigY = sigB_32 + ((expB != 0) ? 0x40000000 : sigB_32);
  }
  return softfloat_normRoundPackToF32(signZ, expZ, sigX - softfloat_shiftRightJam32(sigY, expDiff), mode);
}
```

D.115. f32_add

Returns sum of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_subMagsF32(a, b, mode);
} else {
  return softfloat_addMagsF32(a, b, mode);
}
```

D.116. f32_sub

Returns difference of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
U32 a_xor_b = a ^ b;
```

```
if (signF32UI(a_xor_b) == 1) {
  return softfloat_addMagsF32(a, b, mode);
} else {
  return softfloat_subMagsF32(a, b, mode);
}
```

D.117. i32_to_f32

Converts 32-bit signed integer to 32-bit floating point number

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFFF) == 0) {
  return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32(sign, 0x9C, magnitude_of_A, mode);
```

D.118. ui32_to_f32

Converts 32-bit unsigned integer to 32-bit floating point number

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
if (a == 0) {
  return a;
}
if (a[31] == 1) {
  return softfloat_roundPackToF32(0, 0x9D, a >> 1 | (a & 1), mode);
} else {
  return softfloat_normRoundPackToF32(0, 0x9C, a, mode);
}
```

D.119. f32_to_i32

Converts 32-bit floating point number to a signed 32-bit integer

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  return I32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
```

```
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToI32(sign, sig64, mode);
```

D.120. f32_to_ui32

Converts 32-bit floating point number to an unsigned 32-bit integer

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  return UI32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToUI32(sign, sig64, mode);
```

D.121. 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.122. 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.123. 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.124. 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.125. exception_handling_mode

Returns the target privilege mode that will handle synchronous exception `exception_code`

Return Type	`PrivilegeMode`
Arguments	`ExceptionCode exception_code`

```
if (mode() == PrivilegeMode::M) {
  return PrivilegeMode::M;
} else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::HS) || (mode() == PrivilegeMode::U) {
  if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
    return PrivilegeMode::HS;
  } else {
    return PrivilegeMode::M;
  }
} else {
  assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU, "Unexpected
mode");
  if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
    if (($bits(hedeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
      return PrivilegeMode::VS;
```

```
    } else {
      return PrivilegeMode::HS;
    }
  } else {
    return PrivilegeMode::M;
  }
}
```

D.126. creg2reg

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

Table 12. 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.127. 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.128. 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.129. 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.130. 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.131. 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.132. 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.133. 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.134. raise_guest_page_fault

Raise a guest page fault exception.

Return Type	`void`
Arguments	`MemoryOperation op, XReg gpa, XReg gva, XReg tinst_value, PrivilegeMode from_mode`

```
ExceptionCode code;
Boolean write_gpa_in_tval;
if (op == MemoryOperation::Read) {
  code = ExceptionCode::LoadGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_LOAD_GUEST_PAGE_FAULT;
} else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
  code = ExceptionCode::StoreAmoGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_STORE_AMO_GUEST_PAGE_FAULT;
} else {
  assert(op == MemoryOperation::Fetch, "unexpected memory operation");
  code = ExceptionCode::InstructionGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_INSTRUCTION_GUEST_PAGE_FAULT;
}
PrivilegeMode handling_mode = exception_handling_mode(code);
if (handling_mode == PrivilegeMode::S) {
  htval.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  htinst.VALUE = tinst_value;
  sepc.PC = $pc;
  if (!stval_readonly?()) {
    stval.VALUE = stval_for(code, gva);
  }
  $pc = {stvec.BASE, 2'b00};
  scause.INT = 1'b0;
  scause.CODE = $bits(code);
  hstatus.GVA = 1;
  hstatus.SPV = 1;
  hstatus.SPVP = $bits(from_mode)[0];
  mstatus.SPP = $bits(from_mode)[0];
} else {
  assert(handling_mode == PrivilegeMode::M, "unexpected privilege mode");
  mtval2.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  mtinst.VALUE = tinst_value;
  mstatus.MPP = $bits(from_mode)[1:0];
  if (MXLEN == 64) {
    mstatus.MPV = 1;
  } else {
    mstatush.MPV = 1;
  }
}
set_mode(handling_mode);
abort_current_instruction();
```

D.135. 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.136. raise_precise

Raise synchronous exception number `exception_code`.

Return Type	`void`
Arguments	`ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval`

```
PrivilegeMode handling_mode = exception_handling_mode(exception_code);
if (handling_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
  if (!mtval_readonly?()) {
    mtval.VALUE = mtval_for(exception_code, tval);
  }
  $pc = {mtvec.BASE, 2'b00};
  mcause.INT = 1'b0;
  mcause.CODE = $bits(exception_code);
  if (misa.H == 1) {
    mtval2.VALUE = 0;
    mtinst.VALUE = 0;
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      if (MXLEN == 32) {
        mstatush.MPV = 1;
      } else {
        mstatus.MPV = 1;
      }
    } else {
      if (MXLEN == 32) {
        mstatush.MPV = 0;
      } else {
        mstatus.MPV = 0;
      }
    }
  }
  mstatus.MPP = $bits(from_mode);
} else if (misa.S == 1 && (handling_mode == PrivilegeMode::S)) {
  sepc.PC = $pc;
  if (!stval_readonly?()) {
    stval.VALUE = stval_for(exception_code, tval);
  }
  $pc = {stvec.BASE, 2'b00};
  scause.INT = 1'b0;
  scause.CODE = $bits(exception_code);
  mstatus.SPP = $bits(from_mode)[0];
  if (misa.H == 1) {
    htval.VALUE = 0;
    htinst.VALUE = 0;
    hstatus.SPV = $bits(from_mode)[2];
```

```
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      hstatus.SPV = 1;
      if (exception_code == ExceptionCode::Breakpoint) && (REPORT_VA_IN_STVAL_ON_BREAKPOINT || exception_code ==
ExceptionCode::LoadAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || exception_code ==
ExceptionCode::StoreAmoAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || exception_code ==
ExceptionCode::InstructionAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || exception_code ==
ExceptionCode::LoadAccessFault) && (REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || exception_code ==
ExceptionCode::StoreAmoAccessFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || exception_code ==
ExceptionCode::InstructionAccessFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || exception_code ==
ExceptionCode::LoadPageFault) && (REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || exception_code ==
ExceptionCode::StoreAmoPageFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || exception_code ==
ExceptionCode::InstructionPageFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT) {
        hstatus.GVA = 1;
      } else {
        hstatus.GVA = 0;
      }
      hstatus.SPVP = $bits(from_mode)[0];
    } else {
      hstatus.SPV = 0;
      hstatus.GVA = 0;
    }
  }
} else if (misa.H == 1 && (handling_mode == PrivilegeMode::VS)) {
  vsepc.PC = $pc;
  if (!vstval_readonly?()) {
    vstval.VALUE = vstval_for(exception_code, tval);
  }
  $pc = {vstvec.BASE, 2'b00};
  vscause.INT = 1'b0;
  vscause.CODE = $bits(exception_code);
  vsstatus.SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();
```

D.137. ialign

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

Return Type	`Bits⑥`
Arguments	None

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

D.138. 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.139. 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.140. valid_interrupt_code?

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

Return Type	`Boolean`
Arguments	`XReg code`

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

D.141. valid_exception_code?

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

Return Type	`Boolean`
Arguments	`XReg code`

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

D.142. xlen

Returns the effective XLEN for the current privilege mode.

Return Type	`Bits⑧`

Arguments	None

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

D.143. 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.144. mask_eaddr

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

Return Type	XReg
Arguments	XReg eaddr

```
return eaddr;
```

D.145. pmp_match_64

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 8) * 2;
  Bits<6> shamt = (i % 8) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<64> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_hi = 0;
  Bits<PHYS_ADDR_WIDTH> range_lo = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_lo = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_lo = (pmpaddr_value & ~mask) << 2;
    Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;
    range_hi = pmpaddr_value & ~mask) + len) << 2;   } else if (cfg.A == $bits(PmpCfg_A::NA4 {
    range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
    range_hi = range_lo + 4;
  }
  if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

D.146. 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_hi = 0;
  Bits<PHYS_ADDR_WIDTH> range_lo = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_lo = 0;
```

```
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_lo = (pmpaddr_value & ~mask) << 2;
    Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;
    range_hi = pmpaddr_value & ~mask) + len) << 2;   } else if (cfg.A == $bits(PmpCfg_A::NA4 {
    range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
    range_hi = range_lo + 4;
  }
  if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

D.147. 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.148. mpv

Returns the current value of CSR[mstatus].MPV (when MXLEN == 64) of CSR[mstatush].MPV (when MXLEN == 32)

Return Type	`Bits①`
Arguments	None

```
if (implemented?(ExtensionName::H)) {
  return (MXLEN == 32) ? mstatush.MPV : mstatus.MPV;
} else {
  assert(false, "TODO");
}
```

D.149. effective_ldst_mode

Returns the effective privilege mode for normal explicit loads and stores, taking into account the current actual privilege mode and modifications from mstatus.MPRV.

Return Type	`PrivilegeMode`
Arguments	None

```
if (mode() == PrivilegeMode::M) {
  if (misa.U == 1 && mstatus.MPRV == 1) {
```

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

D.150. 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.151. 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.152. base32?

return True iff current effective XLEN == 32

Return Type	`Boolean`
Arguments	None

```
if (MXLEN == 32) {
  return true;
} else {
  XRegWidth xlen32 = XRegWidth::XLEN32;
  if (mode() == PrivilegeMode::M) {
    return misa.MXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    return mstatus.SXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    return mstatus.UXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    return hstatus.VSXL == $bits(xlen32);
  } else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU, "Unexpected mode");
    return vsstatus.UXL == $bits(xlen32);
  }
}
```

D.153. base64?

return True iff current effective XLEN == 64

Return Type	`Boolean`
Arguments	None

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

D.154. current_translation_mode

Returns the current first-stage translation mode for an explicit load or store from mode given the machine state (e.g., value of satp or vsatp csr).

Returns SatpMode::Reserved if the setting found in satp or vsatp is invalid.

Return Type	`SatpMode`
Arguments	`PrivilegeMode mode`

```
PrivilegeMode effective_mode = effective_ldst_mode();
if (effective_mode == PrivilegeMode::M) {
  return SatpMode::Bare;
}
if (misa.H == 1'b1) {
  if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU) {
    Bits<4> mode_val = vsatp.MODE;
    if (mode_val == $bits(SatpMode::Sv32)) {
      if (MXLEN == 64) {
        if ((effective_mode == PrivilegeMode::VS) && (hstatus.VSXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
```

```
        }
        if ((effective_mode == PrivilegeMode::VU) && (vsstatus.UXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
      }
      if (!SV32_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv32;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV39_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv39;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV48_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv48;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV57_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv57;
    } else {
      return SatpMode::Reserved;
    }
  }
} else if (misa.S == 1'b1) {
  assert(effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U, "unexpected priv mode");
  Bits<4> mode_val = satp.MODE;
  if (mode_val == $bits(SatpMode::Sv32)) {
    if (MXLEN == 64) {
      if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
    }
    if (!implemented?(ExtensionName::Sv32)) {
      return SatpMode::Reserved;
    }
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv39)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv39;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
```

```
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv48)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv48;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv57)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv57;
  } else {
    return SatpMode::Reserved;
  }
}
```

D.155. current_gstage_translation_mode

Returns the current second-stage translation mode for a load or store from VS-mode or VU-mode.

Return Type	`HgatpMode`
Arguments	None

```
return $enum(HgatpMode, hgatp.MODE);
```

D.156. translate_gstage

Translates a guest physical address to a physical address.

Return Type	`TranslationResult`
Arguments	`XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U) {
  result.paddr = gpaddr;
  return result;
}
Boolean mxr = mstatus.MXR == 1;
if (GSTAGE_MODE_BARE && hgatp.MODE == $bits(HgatpMode::Bare)) {
  result.paddr = gpaddr;
  return result;
} else if (SV32X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv32x4)) {
  return gstage_page_walk<32, 34, 32, 2>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV39X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv39x4)) {
  return gstage_page_walk<39, 56, 64, 3>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV48X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv48x4)) {
  return gstage_page_walk<48, 56, 64, 4>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV57X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv57x4)) {
  return gstage_page_walk<57, 56, 64, 5>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else {
  if (op == MemoryOperation::Read) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else {
```

```
    assert(op == MemoryOperation::Fetch, "unexpected memory op");
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  }
}
```

D.157. tinst_value_for_guest_page_fault

Returns the value of htinst/mtinst for a Guest Page Fault

Return Type	`XReg`
Arguments	`MemoryOperation op, Bits<INSTR_ENC_SIZE> encoding, Boolean for_final_vs_pte`

```
if (for_final_vs_pte) {
  if (op == MemoryOperation::Fetch) {
    if (TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else {
      assert(TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always pseudoinstruction", "Instruction guest page
faults can only report zero/pseudo instruction in tval");
      return 0x00002000;
    }
  } else if (op == MemoryOperation::Read) {
    if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002000;
      } else {
        return 0x00003000;
      }
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002020;
      } else {
        return 0x00003020;
      }
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  }
} else {
  if (REPORT_GPA_IN_TVAL_ON_INTERMEDIATE_GUEST_PAGE_FAULT) {
    if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
      return 0x00002000;
    } else if ((VSXLEN == 64) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN64)) {
      return 0x00003000;
    }
  }
}
```

D.158. 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.159. 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.160. 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.161. gstage_page_walk

Translate guest physical address to physical address through a page walk.

May raise a Guest Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated physical address.

Return Type	`TranslationResult`
Arguments	`XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Boolean for_final_vs_pte, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadGuestPageFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionGuestPageFault : ExceptionCode::StoreAmoGuestPageFault);
Boolean mxr = for_final_vs_pte && (mstatus.MXR == 1);
Boolean pbmte = menvcfg.PBMTE == 1;
Boolean adue = menvcfg.ADUE == 1;
Bits<32> tinst = tinst_value_for_guest_page_fault(op, encoding, for_final_vs_pte);
U32 max_gpa_width = LEVELS * VPN_SIZE + 2 + 12;
if (gpaddr >> max_gpa_width != 0) {
  raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
}
ppn = hgatp.PPN;
for (U32 i = (LEVELS - 1); i >= 0; i--) {
  U32 this_vpn_size = (i == (LEVELS - 1)) ? VPN_SIZE + 2 : VPN_SIZE;
  U32 vpn = (gpaddr >> (12 + VPN_SIZE * i)) & 1 << this_vpn_size) - 1);   Bits<PA_SIZE> pte_paddr = (ppn << 12) + (vpn
* (PTESIZE / 8;
  if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_paddr, PTESIZE)) {
    raise(access_fault_code, PrivilegeMode::U, vaddr);
  }
  access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
  Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_paddr);
  PteFlags pte_flags = pte[9:0];
  if ((VA_SIZE != 32) && (pte[60:54] != 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:10] != 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.162. stage1_page_walk

Translate virtual address to physical address through a page walk.

May raise a Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may

raise Access Faults

The translated address *is not* accessed checked.

Returns the translated guest physical address.

Return Type	`TranslationResult`
Arguments	`Bits<MXLEN> vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadPageFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionPageFault : ExceptionCode::StoreAmoPageFault);
Boolean sse = false;
Boolean adue;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  adue = henvcfg.ADUE == 1;
} else {
  adue = menvcfg.ADUE == 1;
}
Boolean pbmte;
if (VA_SIZE == 32) {
  pbmte = false;
} else {
  if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    pbmte = henvcfg.PBMTE == 1;
  } else {
    pbmte = menvcfg.PBMTE == 1;
  }
}
Boolean mxr;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  mxr = (mstatus.MXR == 1) || (vsstatus.MXR == 1);
} else {
  mxr = mstatus.MXR == 1;
}
Boolean sum;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS)) {
  sum = vsstatus.SUM == 1;
} else {
  sum = mstatus.SUM == 1;
}
ppn = vsatp.PPN;
if ((VA_SIZE < xlen()) && (vaddr[xlen() - 1:VA_SIZE] != {xlen() - VA_SIZE{vaddr[VA_SIZE - 1]}})) {
  raise(page_fault_code, effective_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, effective_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[60:54] != 0)) {
    raise(page_fault_code, effective_mode, vaddr);
  }
  if (pte_flags.V == 0) {
    raise(page_fault_code, effective_mode, vaddr);
  }
  if (!sse) {
    if ((pte_flags.R == 0) && (pte_flags.W == 1)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
```

```
  }
  if (pbmte) {
    if (pte[62:61] == 3) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  } else {
    if ((PTESIZE >= 64) && (pte[62:61] != 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  }
  if (!implemented?(ExtensionName::Svnapot)) {
    if ((PTESIZE >= 64) && (pte[63] != 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  }
  if (pte_flags.R == 1 || pte_flags.X == 1) {
    if (op == MemoryOperation::Read || op == MemoryOperation::ReadModifyWrite) {
      if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
        raise(page_fault_code, effective_mode, vaddr);
      }
      if (effective_mode == PrivilegeMode::U && pte_flags.U == 0) {
        raise(page_fault_code, effective_mode, vaddr);
      } else if (misa.H == 1 && effective_mode == PrivilegeMode::VU && pte_flags.U == 0) {
        raise(page_fault_code, effective_mode, vaddr);
      } else if (effective_mode == PrivilegeMode::S && pte_flags.U == 1 && !sum) {
        raise(page_fault_code, effective_mode, vaddr);
      } else if (effective_mode == PrivilegeMode::VS && pte_flags.U == 1 && !sum) {
        raise(page_fault_code, effective_mode, vaddr);
      }
    }
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    } else if ((op == MemoryOperation::Fetch) && ss_page) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    raise(page_fault_code, effective_mode, vaddr) if;
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op ==
MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Write, effective_mode,
encoding);
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_phys.paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
          success = atomic_check_then_write_64(pte_phys.paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
          i = i + 1;
        } else {
          TranslationResult pte_phys = translate_gstage({(pte[PA_SIZE - 3:(i * VPN_SIZE) + 10] << 2), vaddr[11:0]},
vaddr, op, effective_mode, encoding);
          result.paddr = pte_phys.paddr;
          result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
```

```
        raise(page_fault_code, effective_mode, vaddr);
      }
    }
    TranslationResult pte_phys = translate_gstage({(pte[PA_SIZE - 3:(i * VPN_SIZE) + 10] << 2), vaddr[11:0]}, vaddr, op,
effective_mode, encoding);
    result.paddr = pte_phys.paddr;
    if (PTESIZE >= 64) {
      result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
    }
    result.pte_flags = pte_flags;
    return result;
  } else {
    if (i == 0) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
  }
}
```

D.163. 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, effective_mode, vaddr);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise(ExceptionCode::StoreAmoPageFault, effective_mode, vaddr);
  } else {
    assert(op == MemoryOperation::Fetch, "Unexpected memory operation");
    raise(ExceptionCode::InstructionPageFault, effective_mode, vaddr);
  }
}
if (translation_mode == SatpMode::Sv32) {
  result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv39) {
  result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv48) {
```

```
  result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv57) {
  result = stage1_page_walk<57, 56, 64, 5>(vaddr, op, effective_mode, encoding);
} else {
  assert(false, "Unexpected SatpMode");
}
maybe_cache_translation(vaddr, op, result);
return result;
```

D.164. canonical_vaddr?

Returns whether or not *vaddr* is a valid (*i.e.*, canonical) virtual address.

If pointer masking (S**pm) is enabled, then vaddr will be masked before checking the canonical address.

Return Type	`Boolean`
Arguments	`XReg vaddr`

```
if (misa.S == 1'b0) {
  return true;
}
SatpMode satp_mode;
if (virtual_mode?()) {
  satp_mode = $enum(SatpMode, vsatp.MODE);
} else {
  satp_mode = $enum(SatpMode, satp.MODE);
}
XReg eaddr = mask_eaddr(vaddr);
if (satp_mode == SatpMode::Bare) {
  return true;
} else if (satp_mode == SatpMode::Sv32) {
  return true;
} else if (satp_mode == SatpMode::Sv39) {
  return eaddr[63:39] == {25{eaddr[38]}};
} else if (satp_mode == SatpMode::Sv48) {
  return eaddr[63:48] == {16{eaddr[47]}};
} else if (satp_mode == SatpMode::Sv57) {
  return eaddr[63:57] == {6{eaddr[56]}};
}
```

D.165. canonical_gpaddr?

Returns whether or not gpaddr is a valid (*i.e.*, canonical) guest physical address.

Return Type	`Boolean`
Arguments	`XReg gpaddr`

```
SatpMode satp_mode = $enum(SatpMode, satp.MODE);
if (satp_mode == SatpMode::Bare) {
  return true;
} else if (satp_mode == SatpMode::Sv32) {
  return true;
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv39)) {
  return gpaddr[63:39] == {25{gpaddr[38]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv48)) {
  return gpaddr[63:48] == {16{gpaddr[47]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv57)) {
  return gpaddr[63:57] == {6{gpaddr[56]}};
}
```

D.166. 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?<4>(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.167. read_memory_aligned

Read from virtual memory using a known aligned address.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
return read_physical_memory<LEN>(result.paddr);
```

D.168. read_memory

Read from virtual memory.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  return read_memory_aligned<LEN>(virtual_address, encoding);
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with
large atomicity granule");
  physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
```

```
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
    return read_physical_memory<LEN>(physical_address);
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    Bits<LEN> result = 0;
    for (U32 i = 0; i <= (LEN / 8); i++) {
      result = result | (read_memory_aligned<8>(virtual_address + i, encoding) << (8 * i));
    }
    return result;
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior
when any part of the misaligned access causes an exception");
  }
}
```

D.169. 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.170. 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.171. read_memory_xlen_aligned

Read from virtual memory XLEN (which may be runtime-determined) bits using a known aligned address.

Return Type	`Bits<MXLEN>`

Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
  return read_physical_memory<32>(result.paddr);
} else {
  return read_physical_memory<64>(result.paddr);
}
```

D.172. 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.173. 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.174. load_reserved

Register a reservation for virtual_address at least N bits long and read the value from memory.

If aq is set, then also perform a memory model acquire.

If rl is set, then also perform a memory model release (software is discouraged from doing so).

This function assumes alignment checks have already occurred.

Return Type	`Bits<N>`
Arguments	`Bits<MXLEN> virtual_address, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
}
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
register_reservation_set(physical_address, N);
if (misa.S == 1 && LRSC_FAIL_ON_VA_SYNONYM) {
  reservation_virtual_address = virtual_address;
}
return read_memory_aligned<N>(physical_address, encoding);
```

D.175. store_conditional

Atomically check the reservation set to ensure:

- it is valid
- it covers the region addressed by this store
- the address setting the reservation set matches virtual address

If the preceding are met, perform the store and return `0`. Otherwise, return 1.

Return Type	`Boolean`
Arguments	`Bits<MXLEN> virtual_address, Bits<MXLEN> value, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_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.176. 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, effective_ldst_mode(), virtual_address);
}
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::ReadModifyWrite,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::AmoNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_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, effective_ldst_mode(), virtual_address);
} else if (op == AmoOperation::And || op == AmoOperation::Or || op == AmoOperation::Xor && !
pma_applies?(PmaAttribute::AmoLogical, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_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, effective_ldst_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.177. write_memory_aligned

Write to virtual memory using a known aligned address.

Return Type	`void`
Arguments	`XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
write_physical_memory<LEN>(physical_address, value);
```

D.178. write_memory

Write to virtual memory

Return Type	`void`
Arguments	`XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  write_memory_aligned<LEN>(virtual_address, value, encoding);
  return ;
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with
large atomicity granule");
  physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
    write_physical_memory<LEN>(physical_address, value);
    return ;
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    for (U32 i = 0; i <= (LEN / 8); i++) {
      write_memory_aligned<8>(virtual_address + i, (value >> (8 * i))[7:0], encoding);
    }
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior
when any part of the misaligned access causes an exception");
  }
}
```

D.179. write_memory_xlen_aligned

Write to virtual memory XLEN bits (which may be runtime determined) using a known aligned address.

Return Type	`void`
Arguments	`XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
access_check(physical_address, xlen(), virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
  write_physical_memory<32>(physical_address, value);
} else {
  write_physical_memory<64>(physical_address, value);
}
```

D.180. 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.181. 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;
```