MC100-64 Processor Certification Requirements Document

Table of Contents

Revision History

History of documentation changes that eventually lead to releases.

Date	Revision	Changes
2024-07-29	0.7.0	• First version after moving non-microcontroller content in this document to a new document called "RISC-V CRDs (Certification Requirement Documents)" • Change MC100 Unpriv ISA spec from "riscv-spec-v2.1, May 31, 2016" to https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC since the former isn't ratified by the latter is the oldest ratified version. • Added requirements for WFI instruction • Added requirements related to msip memory-mapped register

Date	Revision	Changes
2024-07-11	0.6.0	• Supporting multiple MC versions to support customers wanting to certify existing microcontrollers not using the latest version of ratified standards. • Changed versioning scheme to use major.minor.patch instead of 3-digit major & minor. • Added a table showing the mapping from MC version to ISA manuals. • Reluctantly made interrupts OUT OF SCOPE for MC100 since only the CLINT interrupt controller was ratified at that time and isn't anticipated to be the interrupt controller used by MC100 implementations. • Clarified MANDATORY behaviors for mie and mip CSRs • Removed canonical discovery recipe because the OPT-* options directly inform the certification tests and certification reference model of the status of the various options. Also, canonical discovery recipes (e.g., probing for CLIC) violate the certification approach of avoiding writing potentially illegal values to CSR fields. • Added more options for interrupts • Moved non-microcontroller content in this document to a new document called "RISC-V Certification Plans"
2024-06-03	0.5.0	• Renamed to "RISC-V Microcontroller Certification Plan" based on Jason's recommendation • Added mvendorid, marchid, mimpid, and mhardid read-only priv CSRs because Allen pointed out these are mandatory in M-mode v1.13 (probably older versions too, haven't looked yet). • Added table showing mapping of MC versions to associated RISC-V specifications
2024-06-03	0.4.0	• Added M-mode instruction requirements • Made Zicntr MANDATORY due to very low cost for implementations to support (in the spirit of minimizing options). • Removed OPT-CNTR-PREC since minstret and mcycle must be a full 64 bits to be standard-compliant.
2024-05-25	0.3.0	• Includes Zicntr as OPTIONAL and then has only 32-bit counters for instret and cycle.
2024-05-20	0.2.0	• Very early draft
2024-05-16	0.1.0	• Initial version

Typographic Conventions

CSR field colors

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

CSR field types

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

1. Introduction

The MC100 Processor CRD (Certification Requirements Document) defines the requirements a processor implementation must meet in order to be eligible for the associated MC100 certificate. MC100 is a basic RISC-V processor with minimal M-mode support and has 32-bit and 64-bit variants.

MC100 is not intended for the smallest possible microcontrollers but rather for applications benefiting from a minimal but standardized microcontroller. It consists of:

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

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

1.1. What's a CRD?

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

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

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

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

1.2. CRD Naming Scheme

CRDs have 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 CRD release
  - Format is <major>[.<minor>[.<patch>]]
  - Follows semantic versioning scheme (https://semver.org/)
  - The <major> release is updated when certification test changes are made that **could** cause a previously certified implementation to now fail. Examples are fixing a test bug, or increasing test coverage, or requiring a new version of a standard A <major> release of 0 is used for pre-release versions of a CRD and release versions start with 1.

- The <minor> release is updated when a CRD increases support for optional behaviors. Examples are supporting for new optional standards or supporting additional optional behaviors for standards already in a certificate.
- The <patch> release is updated when certification test changes are made that **can't** cause a previously certified implementation to now fail. Examples are test changes not designed to increase coverage or fixing a documentation typo.
- If omitted, defaults to v1.0.0
- Examples: v1, v1.1, v2.3.1, 0.3.4 (pre-release)

1.3. CRD Terminology

Table 1. 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 2. Glossary

Term	Meaning
CRD	Certification Requirements Document
N/A	"Not Applicable"
AKA	"Also Known As"

1.4. Processor CRDs

There are Processor CRDs for different classes of RISC-V processors. These documents augment information in the related TSC Profile when available and/or other RVI standards documents (e.g., Priv and Unpriv ISA manuals). Only ratified extensions are candidates for certification. This implies all custom extensions are also OUT-OF-SCOPE.

1.4.1. Processor CRD Naming Scheme

Processor CRD names have the following format:

```
<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 a CRD supports multiple bases and <base> is omitted in a reference, it applies to all supported bases
  - If a CRD only supports one base then <base> is generally omitted

CRD	TSC Profile	Description
MC100-series	TBD	32/64-bit minimal microcontroller that runs low-level software on an RTOS or bare-metal (no virtual memory)
MC200-series	TBD	32/64-bit intermediate microcontroller
MC300-series	TBD	32/64-bit advanced microcontroller
AC100-series	RVB23	64-bit Apps-processor running Bespoke rich operating systems (e.g., Yocto Linux)
AC200-series	RVA23	64-bit Apps-processor running standard rich operating systems (e.g., commercial Linux distributions, Android)

1.4.2. CSR Field Terminology

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 designed to be comprehensive verification test suites anyways.

1.5. Related Specifications

Certificate Model	TSC Profile	Unpriv ISA Manual	Priv ISA Manual	Debug Manual
MC100-64	No profile	20191213	20190608-Priv-MSU-Ratified	0.13.2

1.6. Privileged Modes

M	S	U	VS	VU
MANDATORY	OUT-OF-SCOPE	OUT-OF-SCOPE	OUT-OF-SCOPE	OUT-OF-SCOPE

2. Extensions

Any RISC-V extensions not listed in this section are OUT-OF-SCOPE. The MC100-64 certificate doesn't cover their behaviors.

2.1. Mandatory Extensions

Requirement ID	Extension	Version	Long Name	Note
REQ-EXT-C	C	~> 2.2	Compressed instructions	
REQ-EXT-I	I	~> 2.1	Base integer ISA (RV32I or RV64I)	
REQ-EXT-M	M	~> 2.0	Integer multiply and divide instructions	
REQ-EXT-Sm	Sm	~> 1.11.0	Machine mode	
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.

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	Any	Sm	
MTVEC_BASE_ALIGNMENT_VECTORED	≥ 4	Any	Sm	
MTVEC_MODES	1-element to 2-element array of [0, 1]	Any	Sm	
M_MODE_ENDIANESS	[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	

Parameter	Type	Allowed Value(s)	Extension(s)	Note
VENDOR_ID_OFFSET	7-bit integer	Any	Sm	
XLEN	[32, 64]	64	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.

Parameters	Type	Extension(s)
CONFIG_PTR_ADDRESS	integer	Sm
MUTABLE_MISA_C	boolean	C
MUTABLE_MISA_M	boolean	M
PMA_GRANULARITY	2 to 66	Sm
REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION	boolean	Sm
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	Sm
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	Sm
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	Sm
TRAP_ON_ILLEGAL_WLRL	boolean	Sm
TRAP_ON_RESERVED_INSTRUCTION	boolean	Sm
TRAP_ON_UNIMPLEMENTED_CSR	boolean	Sm
TRAP_ON_UNIMPLEMENTED_INSTRUCTION	boolean	Sm

4. Traps

RISC-V supports both synchronous exceptions and asynchronous interrupts. TODO: List only traps that exist in this certificate model (currently lists all possible in present extensions). See https://github.com/riscv-software-src/riscv-unified-db/issues/291 and https://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
10	Environment call from VS-mode
11	Environment call from M-mode
12	Instruction page fault
13	Load page fault
15	Store/AMO page fault
18	Software Check
20	Instruction guest page fault
21	Load guest page fault
22	Virtual instruction
23	Store/AMO guest 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

5. Instruction Summary

TODO: List only instructions that exist in this certificate model. Currently lists all possible in present extensions so the I extension is providing both RV32I and RV64I instructions. See https://github.com/riscv-software-src/riscv-unified-db/issues/291 and https://github.com/riscv-software-src/riscv-unified-db/issues/324

Name	Long Name
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
c.add	Add
c.addi	Add a sign-extended non-zero immediate
c.addi16sp	Add a sign-extended non-zero immediate
c.addi4spn	Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer
c.addiw	Add a sign-extended non-zero immediate
c.addw	Add word
c.and	And
c.andi	And immediate
c.beqz	Branch if Equal Zero

Name	Long Name
c.bnez	Branch if NOT Equal Zero
c.ebreak	Breakpoint exception.
c.j	Jump
c.jal	Jump and Link
c.jalr	Jump and Link Register.
c.jr	Jump Register
c.ld	Load double
c.ldsp	Load doubleword from stack pointer
c.li	Load the sign-extended 6-bit immediate
c.lui	Load the non-zero 6-bit immediate field into bits 17-12 of the destination register
c.lw	Load word
c.lwsp	Load word from stack pointer
c.mv	Move Register
c.nop	Non-operation
c.or	Or
c.sd	Store double
c.sdsp	Store doubleword to stack
c.slli	Shift left logical immediate
c.srai	Shift right arithmetical immediate
c.srli	Shift right logical immediate
c.sub	Subtract
c.subw	Subtract word
c.sw	Store word
c.swsp	Store word to stack
c.xor	Exclusive Or
csrrc	No synopsis available.
csrrci	No synopsis available.
csrrs	Atomic Read and Set Bits in CSR
csrrsi	No synopsis available.
csrrw	Atomic Read/Write CSR
csrrwi	Atomic Read/Write CSR Immediate
div	Signed division
divu	Unsigned division

Name	Long Name
divuw	Unsigned 32-bit division
divw	Signed 32-bit division
ebreak	Breakpoint exception
ecall	Environment call
fence	Memory ordering fence
jal	Jump and link
jalr	Jump and link register
lb	Load byte
lbu	Load byte unsigned
ld	Load doubleword
lh	Load halfword
lhu	Load halfword unsigned
lui	Load upper immediate
lw	Load word
lwu	Load word unsigned
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
ori	Or immediate
rem	Signed remainder
remu	Unsigned remainder
remuw	Unsigned 32-bit remainder
remw	Signed 32-bit remainder
sb	Store byte
sd	Store doubleword
sh	Store halfword
sll	Shift left logical
slli	Shift left logical immediate
slliw	Shift left logical immediate word

Name	Long Name
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
wfi	Wait for interrupt
xor	Exclusive Or
xori	Exclusive Or immediate

6. CSR Summary

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
mconfigptr	Machine Configuration Pointer	0xf15	M	Sm >= 1.12

Name	Long Name	Address	Mode	Primary Extension
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
mepc	Machine Exception Program Counter	0x341	M	Sm ~> 1.11.0
mhartid	Machine Hart ID	0xf14	M	Sm ~> 1.11.0
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	0xb02	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
mseccfg	Machine Security Configuration	0x747	M	Sm >= 1.12
mseccfgh	Most significant 32 bits of Machine Security Configuration	0x757	M	Sm >= 1.12
mstatus	Machine Status	0x300	M	Sm ~> 1.11.0
mstatush	Machine Status High	0x310	M	Sm >= 1.12
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
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
0x300	M	mstatus	Machine Status	Sm ~> 1.11.0
0x301	M	misa	Machine ISA Control	Sm ~> 1.11.0
0x304	M	mie	Machine Interrupt Enable	Sm ~> 1.11.0
0x305	M	mtvec	Machine Trap Vector Control	Sm ~> 1.11.0
0x310	M	mstatush	Machine Status High	Sm >= 1.12

Address	Mode	Name	Long Name	Primary Extension
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
0x741	M	mnepc	Machine Exception Program Counter	Sm ~> 1.11.0
0x747	M	mseccfg	Machine Security Configuration	Sm >= 1.12
0x757	M	mseccfgh	Most significant 32 bits of Machine Security Configuration	Sm >= 1.12
0xb00	M	mcycle	Machine Cycle Counter	Zicntr ~> 2.0.0
0xb02	M	minstret	Machine Instructions Retired Counter	Zicntr ~> 2.0.0
0xb02	M	minstreth	Machine Instructions Retired Counter	Zicntr ~> 2.0.0
0xb80	M	mcycleh	High-half machine Cycle 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
0xf15	M	mconfigptr	Machine Configuration Pointer	Sm >= 1.12

Appendix A: Extension Details

A.1. Extension C

Long Name: Compressed instructions

Version Requirement: ~> 2.2

2.0.0

State

ratified

Ratification date

2019-12

A.1.1. 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.1.2. 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.

NOTE	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.

TIP

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.

NOTE

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.

TIP

Variable-length instruction sets have long been used to improve code density. For

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

In 1963, CDC introduced the Cray-designed CDC 6600 cite:[cdc6600], a precursor to RISC architectures, that introduced a register-rich load-store architecture with instructions of two lengths, 15-bits and 30-bits. The later Cray-1 design used a very similar instruction format, with 16-bit and 32-bit instruction lengths.

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

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

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

Benefiting from 25 years of hindsight, RISC-V was designed to support compressed instructions from the outset, leaving enough opcode space for RVC to be added as a simple extension on top of the base ISA (along with many other extensions). The philosophy of RVC is to reduce code size for embedded applications *and* to improve

> performance and energy-efficiency for all applications due to fewer misses in the instruction cache. Waterman shows that RVC fetches 25%-30% fewer instruction bits, which reduces instruction cache misses by 20%-25%, or roughly the same performance impact as doubling the instruction cache size. cite:[waterman-ms]

A.1.3. Compressed Instruction Formats

Compressed 16-bit RVC instruction formats 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. Registers specified by the three-bit *rs1'*, *rs2'*, and *rd'* fields of the CIW, CL, CS, CA, and CB formats. 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.

NOTE 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`.

NOTE *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.

NOTE 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.1.4. Instructions

The following instructions are added by this extension:

c.add	**Add**
c.addi	**Add a sign-extended non-zero immediate**
c.addi16sp	**Add a sign-extended non-zero immediate**
c.addi4spn	**Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer**
c.addiw	**Add a sign-extended non-zero immediate**
c.addw	**Add word**
c.and	**And**
c.andi	**And immediate**
c.beqz	**Branch if Equal Zero**
c.bnez	**Branch if NOT Equal Zero**
c.ebreak	**Breakpoint exception.**
c.j	**Jump**
c.jal	**Jump and Link**

c.jalr	**Jump and Link Register.**
c.jr	**Jump Register**
c.ld	**Load double**
c.ldsp	**Load doubleword from stack pointer**
c.li	**Load the sign-extended 6-bit immediate**
c.lui	**Load the non-zero 6-bit immediate field into bits 17-12 of the destination register**
c.lw	**Load word**
c.lwsp	**Load word from stack pointer**
c.mv	**Move Register**
c.nop	**Non-operation**
c.or	**Or**
c.sd	**Store double**
c.sdsp	**Store doubleword to stack**
c.slli	**Shift left logical immediate**
c.srai	**Shift right arithmetical immediate**
c.srli	**Shift right logical immediate**
c.sub	**Subtract**
c.subw	**Subtract word**
c.sw	**Store word**
c.swsp	**Store word to stack**
c.xor	**Exclusive Or**

A.1.5. OUT-OF-SCOPE Parameters

MUTABLE_MISA_C ⇒ boolean

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

A.2. 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.2.1. Synopsis

Base integer instructions — TODO

A.2.2. Instructions

The following instructions are added by this extension:

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	**Memory ordering fence**
jal	**Jump and link**
jalr	**Jump and link register**
lb	**Load byte**
lbu	**Load byte unsigned**
ld	**Load doubleword**
lh	**Load halfword**
lhu	**Load halfword unsigned**
lui	**Load upper immediate**

lw	**Load word**
lwu	**Load word unsigned**
or	**Or**
ori	**Or immediate**
sb	**Store byte**
sd	**Store doubleword**
sh	**Store halfword**
sll	**Shift left logical**
slli	**Shift left logical immediate**
slliw	**Shift left logical immediate word**
sllw	**Shift left logical word**
slt	**Set on less than**
slti	**Set on less than immediate**
sltiu	**Set on less than immediate unsigned**
sltu	**Set on less than unsigned**
sra	**Shift right arithmetic**
srai	**Shift right arithmetic immediate**
sraiw	**Shift right arithmetic immediate word**
sraw	**Shift right arithmetic word**
srl	**Shift right logical**
srli	**Shift right logical immediate**
srliw	**Shift right logical immediate word**
srlw	**Shift right logical word**
sub	**Subtract**
subw	**Subtract word**
sw	**Store word**
xor	**Exclusive Or**
xori	**Exclusive Or immediate**

A.3. Extension M

Long Name: Integer multiply and divide instructions

Version Requirement: ~> 2.0

2.0.0

State

ratified

Ratification date

2019-12

A.3.1. 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.

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

A.3.2. Instructions

The following instructions are added by this extension:

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.3.3. OUT-OF-SCOPE Parameters

MUTABLE_MISA_M ⇒ boolean

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

A.4. 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.4.1. 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.4.2. Instructions

The following instructions are added by this extension:

mret	**Machine Exception Return**
wfi	**Wait for interrupt**

A.4.3. 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.

NOTE	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

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.

M_MODE_ENDIANESS ⇒ [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

XLEN ⇒ [32, 64]

XLEN in M-mode (AKA MXLEN, tracked by issue #52)

A.4.4. OUT-OF-SCOPE Parameters

CONFIG_PTR_ADDRESS ⇒ 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_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_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_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_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.

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`.

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.5. Extension Zicntr

Long Name: Architectural performance counters

Version Requirement: ~> 2.0

2.0.0

State

ratified

Ratification date

2019-12

A.5.1. Synopsis

Architectural performance counters

A.5.2. 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.6. Extension Zicsr

Long Name: Control and status registers

Version Requirement: ~> 2.0

2.0.0

State

ratified

Ratification date

==== Synopsis

Control and status registers

A.6.1. Instructions

The following instructions are added by this extension:

csrrc	**No synopsis available.**
csrrci	**No synopsis available.**
csrrs	**Atomic Read and Set Bits in CSR**
csrrsi	**No synopsis available.**
csrrw	**Atomic Read/Write CSR**
csrrwi	**Atomic Read/Write CSR Immediate**

Appendix B: Instruction Details

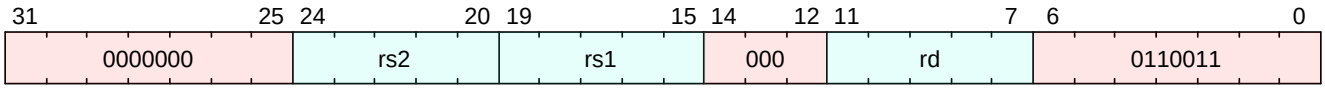
B.1. add

Integer add

This instruction is defined by:

- I, version >= 0

B.1.1. Encoding



B.1.2. Synopsis

Add the value in rs1 to rs2, and store the result in rd. Any overflow is thrown away.

B.1.3. Access

M	S	U
Always	Always	Always

B.1.4. Decode Variables

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

B.1.5. Execution

```
X[rd] = X[rs1] + X[rs2];
```

B.1.6. Exceptions

This instruction does not generate synchronous exceptions.

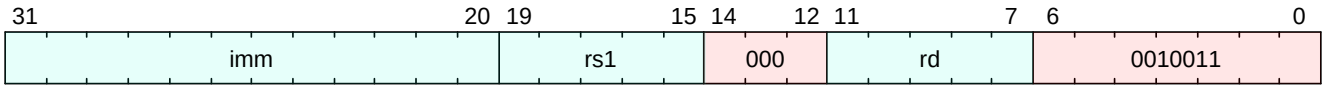
B.2. addi

Add immediate

This instruction is defined by:

- I, version >= 0

B.2.1. Encoding



B.2.2. Synopsis

Add an immediate to the value in rs1, and store the result in rd

B.2.3. Access

M	S	U
Always	Always	Always

B.2.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.2.5. Execution

```
X[rd] = X[rs1] + imm;
```

B.2.6. Exceptions

This instruction does not generate synchronous exceptions.

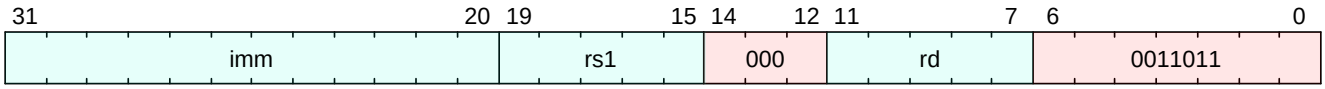
B.3. addiw

Add immediate word

This instruction is defined by:

- I, version >= 0

B.3.1. Encoding



B.3.2. Synopsis

Add an immediate to the 32-bit value in rs1, and store the sign extended result in rd

B.3.3. Access

M	S	U
Always	Always	Always

B.3.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.3.5. Execution

```
XReg operand = sext(X[rs1], 31);
X[rd] = sext(operand + imm, 31);
```

B.3.6. Exceptions

This instruction does not generate synchronous exceptions.

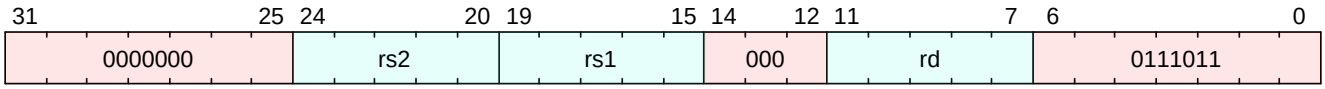
B.4. addw

Add word

This instruction is defined by:

- I, version >= 0

B.4.1. Encoding



B.4.2. Synopsis

Add the 32-bit values in rs1 to rs2, and store the sign-extended result in rd. Any overflow is thrown away.

B.4.3. Access

M	S	U
Always	Always	Always

B.4.4. Decode Variables

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

B.4.5. Execution

```
XReg operand1 = sext(X[rs1], 31);
XReg operand2 = sext(X[rs2], 31);
X[rd] = sext(operand1 + operand2, 31);
```

B.4.6. Exceptions

This instruction does not generate synchronous exceptions.

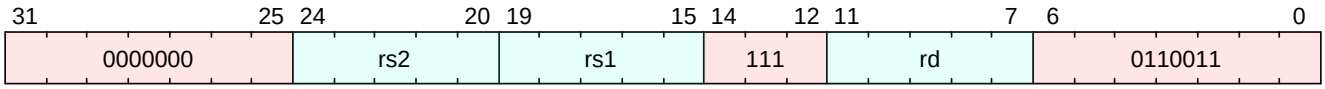
B.5. and

And

This instruction is defined by:

- I, version >= 0

B.5.1. Encoding



B.5.2. Synopsis

And rs1 with rs2, and store the result in rd

B.5.3. Access

M	S	U
Always	Always	Always

B.5.4. Decode Variables

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

B.5.5. Execution

```
X[rd] = X[rs1] & X[rs2];
```

B.5.6. Exceptions

This instruction does not generate synchronous exceptions.

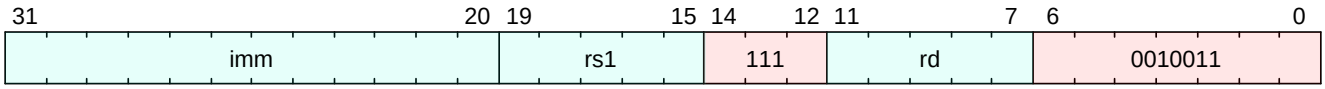
B.6. andi

And immediate

This instruction is defined by:

- I, version >= 0

B.6.1. Encoding



B.6.2. Synopsis

And an immediate to the value in rs1, and store the result in rd

B.6.3. Access

M	S	U
Always	Always	Always

B.6.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.6.5. Execution

```
X[rd] = X[rs1] & imm;
```

B.6.6. Exceptions

This instruction does not generate synchronous exceptions.

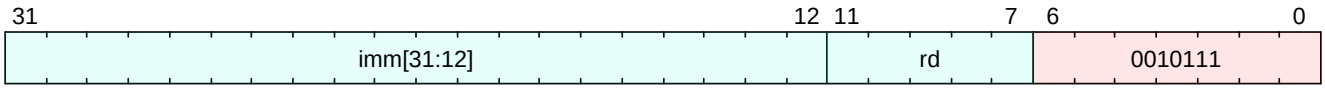
B.7. auipc

Add upper immediate to pc

This instruction is defined by:

- I, version >= 0

B.7.1. Encoding



B.7.2. Synopsis

Add an immediate to the current PC.

B.7.3. Access

M	S	U
Always	Always	Always

B.7.4. Decode Variables

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

B.7.5. Execution

```
X[rd] = $pc + imm;
```

B.7.6. Exceptions

This instruction does not generate synchronous exceptions.

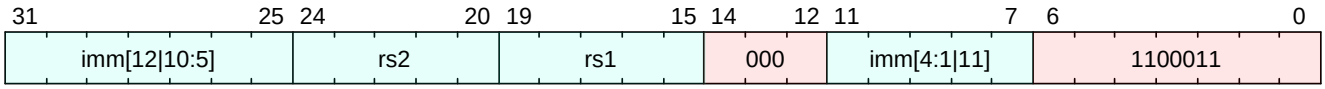
B.8. beq

Branch if equal

This instruction is defined by:

- I, version >= 0

B.8.1. Encoding



B.8.2. Synopsis

Branch to PC + imm if the value in register rs1 is equal to the value in register rs2.

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

B.8.3. Access

M	S	U
Always	Always	Always

B.8.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.8.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs == rhs) {
  jump_halfword($pc + imm);
}
```

B.8.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

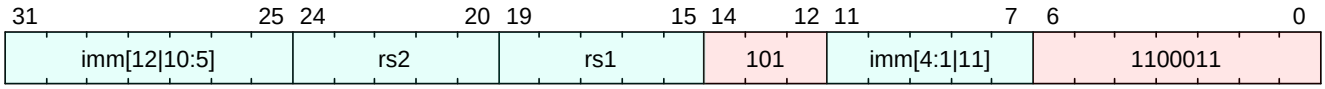
B.9. bge

Branch if greater than or equal

This instruction is defined by:

- I, version >= 0

B.9.1. Encoding



B.9.2. Synopsis

Branch to PC + imm if the signed value in register rs1 is greater than or equal to the signed value in register rs2.

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

B.9.3. Access

M	S	U
Always	Always	Always

B.9.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.9.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if ($signed(lhs) >= $signed(rhs)) {
  jump_halfword($pc + imm);
}
```

B.9.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

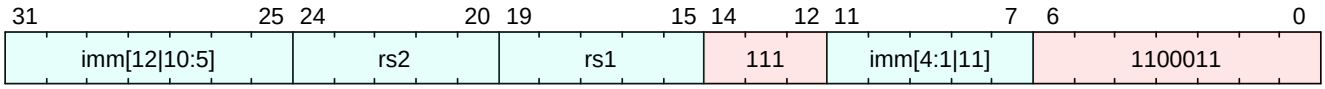
B.10. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

- I, version >= 0

B.10.1. Encoding



B.10.2. Synopsis

Branch to PC + imm if the unsigned value in register rs1 is greater than or equal to the unsigned value in register rs2.

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

B.10.3. Access

M	S	U
Always	Always	Always

B.10.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.10.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs >= rhs) {
  jump_halfword($pc + imm);
}
```

B.10.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

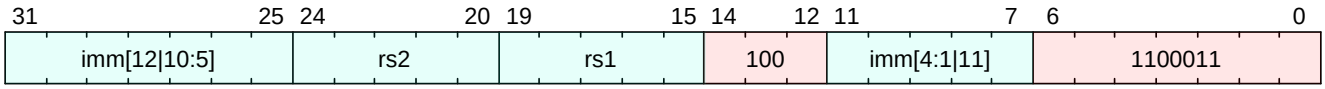
B.11. blt

Branch if less than

This instruction is defined by:

- I, version >= 0

B.11.1. Encoding



B.11.2. Synopsis

Branch to PC + imm if the signed value in register rs1 is less than the signed value in register rs2.

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

B.11.3. Access

M	S	U
Always	Always	Always

B.11.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.11.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if ($signed(lhs) < $signed(rhs)) {
  jump_halfword($pc + imm);
}
```

B.11.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

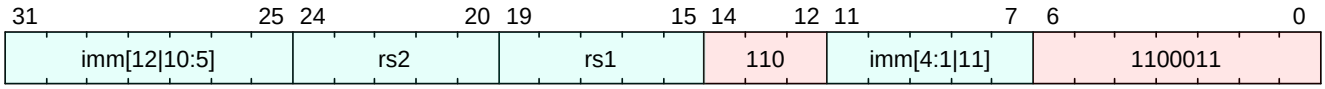
B.12. bltu

Branch if less than unsigned

This instruction is defined by:

- I, version >= 0

B.12.1. Encoding



B.12.2. Synopsis

Branch to PC + imm if the unsigned value in register rs1 is less than the unsigned value in register rs2.

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

B.12.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs < rhs) {
  jump_halfword($pc + imm);
}
```

B.12.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

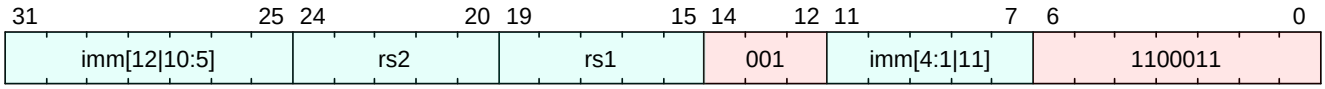
B.13. bne

Branch if not equal

This instruction is defined by:

- I, version >= 0

B.13.1. Encoding



B.13.2. Synopsis

Branch to PC + imm if the value in register rs1 is not equal to the value in register rs2.

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

B.13.3. Access

M	S	U
Always	Always	Always

B.13.4. Decode Variables

```
Bits<13> imm = {$encoding[31], $encoding[7], $encoding[30:25], $encoding[11:8], 1'd0};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.13.5. Execution

```
XReg lhs = X[rs1];
XReg rhs = X[rs2];
if (lhs != rhs) {
  jump_halfword($pc + imm);
}
```

B.13.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

B.14. c.add

Add

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.14.1. Encoding

B.14.2. Synopsis

Add the value in rs2 to rd, and store the result in rd. C.ADD expands into `add rd, rd, rs2`.

B.14.3. Access

M	S	U
Always	Always	Always

B.14.4. Decode Variables

```
Bits<5> rs2 = $encoding[6:2];
Bits<5> rd = $encoding[11:7];
```

B.14.5. Execution

```
XReg t0 = X[rd];
XReg t1 = X[rs2];
X[rd] = t0 + t1;
```

B.14.6. Exceptions

This instruction does not generate synchronous exceptions.

B.15. c.addi

Add a sign-extended non-zero immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.15.1. Encoding

B.15.2. Synopsis

C.ADDI adds the non-zero sign-extended 6-bit immediate to the value in register rd then writes the result to rd. C.ADDI expands into <code>addi rd, rd, imm</code>. C.ADDI is only valid when rd ≠ x0 and imm ≠ 0. The code points with rd=x0 encode the C.NOP instruction; the remaining code points with imm=0 encode HINTs.

B.15.3. Access

M	S	U
Always	Always	Always

B.15.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.15.5. Execution

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

B.15.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

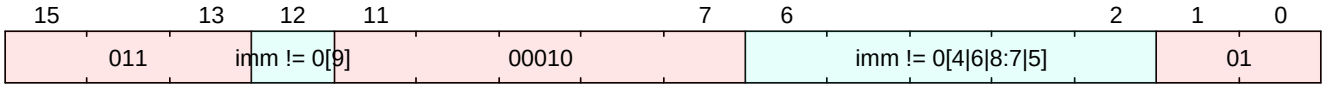
B.16. c.addi16sp

Add a sign-extended non-zero immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.16.1. Encoding



B.16.2. Synopsis

C.ADDI16SP adds the non-zero sign-extended 6-bit immediate to the value in the stack pointer (sp=x2), where the immediate is scaled to represent multiples of 16 in the range (-512,496). C.ADDI16SP is used to adjust the stack pointer in procedure prologues and epilogues. It expands into <code>addi x2, x2, nzimm[9:4]</code>. C.ADDI16SP is only valid when nzimm ≠ 0; the code point with nzimm=0 is reserved.

B.16.3. Access

M	S	U
Always	Always	Always

B.16.4. Decode Variables

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

B.16.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[2] = X[2] + imm;
```

B.16.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

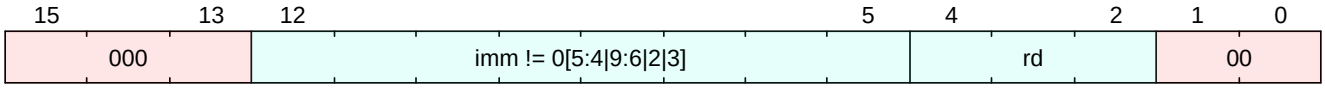
B.17. c.addi4spn

Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.17.1. Encoding



B.17.2. Synopsis

Adds a zero-extended non-zero immediate, scaled by 4, to the stack pointer, x2, and writes the result to rd'. This instruction is used to generate pointers to stack-allocated variables. It expands to <code>addi rd', x2, nzuimm[9:2]</code>. C.ADDI4SPN is only valid when nzuimm ≠ 0; the code points with nzuimm=0 are reserved.

B.17.3. Access

M	S	U
Always	Always	Always

B.17.4. Decode Variables

```
Bits<10> imm = {$encoding[10:7], $encoding[12:11], $encoding[5], $encoding[6], 2'd0};
Bits<3> rd = $encoding[4:2];
```

B.17.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd + 8] = X[2] + imm;
```

B.17.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.18. c.addiw

Add a sign-extended non-zero immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.18.1. Encoding

B.18.2. Synopsis

C.ADDIW is an RV64C/RV128C-only instruction that performs the same computation as C.ADDI but produces a 32-bit result, then sign-extends result to 64 bits. C.ADDIW expands into <code>addiw rd, rd, imm</code>. The immediate can be zero for C.ADDIW, where this corresponds to <code>sext.w rd</code>. C.ADDIW is only valid when rd ≠ x0; the code points with rd=x0 are reserved.

B.18.3. Access

M	S	U
Always	Always	Always

B.18.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.18.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[rd] = sext((X[rd] + imm), 32);
```

B.18.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

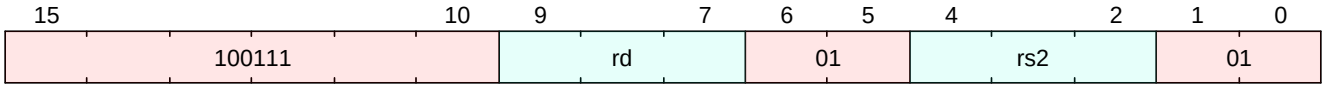
B.19. c.addw

Add word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.19.1. Encoding



B.19.2. Synopsis

Add the 32-bit values in rs2 from rd, and store the result in rd. The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.ADDW expands into `addw rd, rd, rs2`.

B.19.3. Access

M	S	U
Always	Always	Always

B.19.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.19.5. Execution

```
Bits<32> t0 = X[rd + 8][31:0];
Bits<32> t1 = X[rs2 + 8][31:0];
X[rd + 8] = sext(t0 + t1, 31);
```

B.19.6. Exceptions

This instruction does not generate synchronous exceptions.

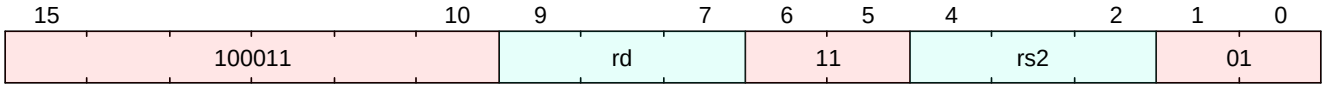
B.20. c.and

And

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.20.1. Encoding



B.20.2. Synopsis

And rd with rs2, and store the result in rd The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.AND expands into `and rd, rd, rs2`.

B.20.3. Access

M	S	U
Always	Always	Always

B.20.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.20.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 & t1;
```

B.20.6. Exceptions

This instruction does not generate synchronous exceptions.

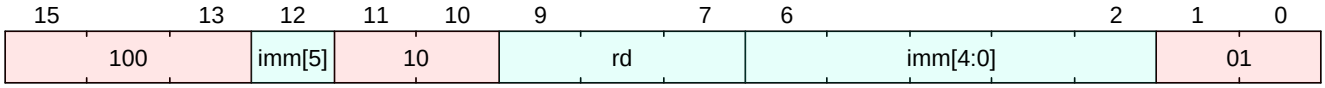
B.21. c.andi

And immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.21.1. Encoding



B.21.2. Synopsis

And an immediate to the value in rd, and store the result in rd. The rd register index should be used as rd+8 (registers x8-x15). C.ANDI expands into `andi rd, rd, imm`.

B.21.3. Access

M	S	U
Always	Always	Always

B.21.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<3> rd = $encoding[9:7];
```

B.21.5. Execution

```
X[rd + 8] = X[rd + 8] & imm;
```

B.21.6. Exceptions

This instruction does not generate synchronous exceptions.

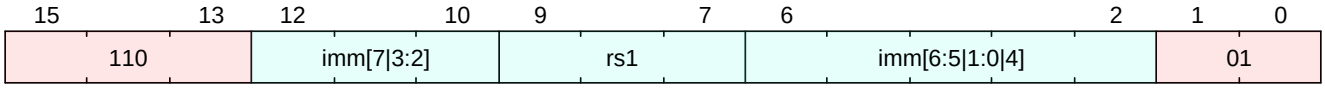
B.22. c.beqz

Branch if Equal Zero

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.22.1. Encoding



B.22.2. Synopsis

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register rs1' is zero. It expands to beq <code>rs1, x0, offset</code>.

B.22.3. Access

M	S	U
Always	Always	Always

B.22.4. Decode Variables

```
Bits<8> imm = {$encoding[12], $encoding[6:5], $encoding[2], $encoding[11:10],
$encoding[4:3]};
Bits<3> rs1 = $encoding[9:7];
```

B.22.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (X[rs1] != 0) {
  jump($pc + imm);
}
```

B.22.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

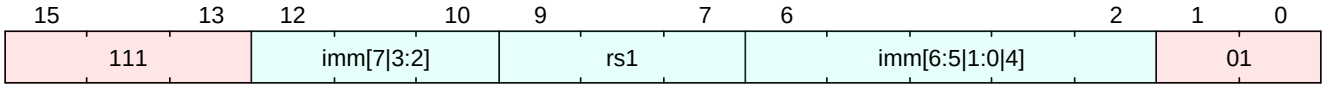
B.23. c.bnez

Branch if NOT Equal Zero

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.23.1. Encoding



B.23.2. Synopsis

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register rs1' is NOT zero. It expands to beq <code>rs1, x0, offset</code>.

B.23.3. Access

M	S	U
Always	Always	Always

B.23.4. Decode Variables

```
Bits<8> imm = {$encoding[12], $encoding[6:5], $encoding[2], $encoding[11:10],
$encoding[4:3]};
Bits<3> rs1 = $encoding[9:7];
```

B.23.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (X[rs1] != 0) {
  jump($pc + imm);
}
```

B.23.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

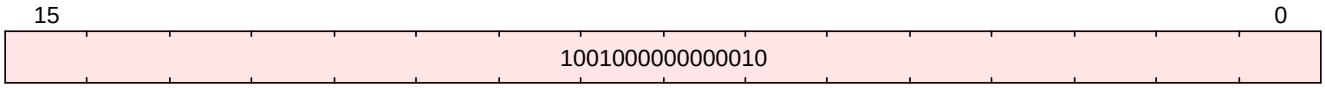
B.24. c.ebreak

Breakpoint exception.

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.24.1. Encoding



B.24.2. Synopsis

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

NOTE	As described in the C Standard 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.24.3. Access

M	S	U
Always	Always	Always

B.24.4. Decode Variables

```
```

B.24.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (TRAP_ON_EBREAK) {
  raise_precise(ExceptionCode::Breakpoint, mode(), $pc);
} else {
  eei_ebreak();
```

```
}
```

B.24.6. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint
- IllegalInstruction

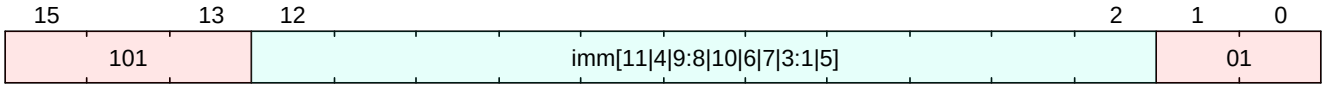
B.25. c.j

Jump

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.25.1. Encoding



B.25.2. Synopsis

C.J performs an unconditional control transfer. The offset is sign-extended and added to the pc to form the jump target address. C.J can therefore target a ±2 KiB range. It expands to jal <code>x0, offset</code>.

B.25.3. Access

M	S	U
Always	Always	Always

B.25.4. Decode Variables

```
signed Bits<12> imm = sext({$encoding[12], $encoding[8], $encoding[10:9],
$encoding[6], $encoding[7], $encoding[2], $encoding[11], $encoding[5:3], 1'd0});
```

B.25.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
jump($pc + imm);
```

B.25.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

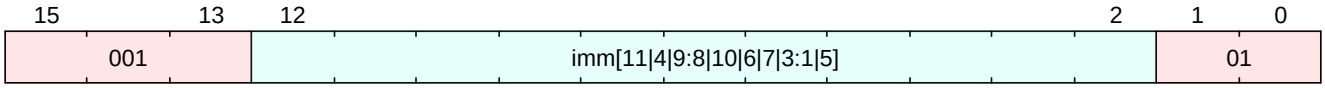
B.26. c.jal

Jump and Link

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.26.1. Encoding



B.26.2. Synopsis

C.JAL is an RV32C-only instruction that performs the same operation as C.J, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. It expands to `jal x1, offset`.

B.26.3. Access

M	S	U
Always	Always	Always

B.26.4. Decode Variables

```
signed Bits<12> imm = sext({$encoding[12], $encoding[8], $encoding[10:9],
$encoding[6], $encoding[7], $encoding[2], $encoding[11], $encoding[5:3], 1'd0});
```

B.26.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg retrun_addr = $pc + 2;
jump_halfword($pc + imm);
X[1] = retrun_addr;
```

B.26.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- InstructionAddressMisaligned

B.27. c.jalr

Jump and Link Register.

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.27.1. Encoding

B.27.2. Synopsis

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. C.JALR expands to jalr x1, 0(rs1).

B.27.3. Access

M	S	U
Always	Always	Always

B.27.4. Decode Variables

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

B.27.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg returnaddr;
returnaddr = $pc + 2;
jump(X[rs1]);
X[1] = returnaddr;
```

B.27.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- InstructionAddressMisaligned

B.28. c.jr

Jump Register

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.28.1. Encoding

B.28.2. Synopsis

C.JR (jump register) performs an unconditional control transfer to the address in register rs1. C.JR expands to jalr x0, 0(rs1).

B.28.3. Access

M	S	U
Always	Always	Always

B.28.4. Decode Variables

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

B.28.5. Execution

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

B.28.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

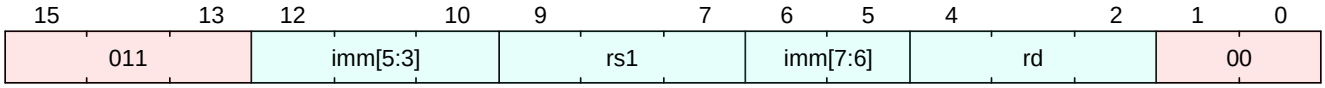
B.29. c.ld

Load double

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.29.1. Encoding



B.29.2. Synopsis

Loads a 64-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1. It expands to `ld rd, offset(rs1)`.

B.29.3. Access

M	S	U
Always	Always	Always

B.29.4. Decode Variables

```
Bits<8> imm = {$encoding[6:5], $encoding[12:10], 3'd0};
Bits<3> rd = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.29.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<64>(virtual_address, $encoding), 64);
```

B.29.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault
- StoreAmoAccessFault

B.30. c.ldsp

Load doubleword from stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.30.1. Encoding

B.30.2. Synopsis

C.LDSP is an RV64C/RV128C-only instruction that loads a 64-bit value from memory into register rd. It computes its effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to ld <code>rd, offset(x2)</code>. C.LDSP is only valid when rd ≠ x0 the code points with rd=x0 are reserved.

B.30.3. Access

M	S	U
Always	Always	Always

B.30.4. Decode Variables

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

B.30.5. Execution

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

B.30.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

B.31. c.li

Load the sign-extended 6-bit immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.31.1. Encoding

B.31.2. Synopsis

C.LI loads the sign-extended 6-bit immediate, imm, into register rd. C.LI expands into <code>addi rd, x0, imm</code>. C.LI is only valid when rd ≠ x0; the code points with rd=x0 encode HINTs.

B.31.3. Access

M	S	U
Always	Always	Always

B.31.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.31.5. Execution

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

B.31.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

B.32. c.lui

Load the non-zero 6-bit immediate field into bits 17-12 of the destination register

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.32.1. Encoding

B.32.2. Synopsis

C.LUI loads the non-zero 6-bit immediate field into bits 17-12 of the destination register, clears the bottom 12 bits, and sign-extends bit 17 into all higher bits of the destination. C.LUI expands into <code>lui rd, imm</code>. C.LUI is only valid when rd≠x0 and rd≠x2, and when the immediate is not equal to zero. The code points with imm=0 are reserved; the remaining code points with rd=x0 are HINTs; and the remaining code points with rd=x2 correspond to the C.ADDI16SP instruction

B.32.3. Access

M	S	U
Always	Always	Always

B.32.4. Decode Variables

```
Bits<18> imm = {$encoding[12], $encoding[6:2], 12'd0};
Bits<5> rd = $encoding[11:7];
```

B.32.5. Execution

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

B.32.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

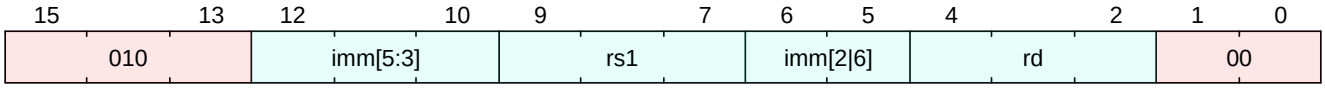
B.33. c.lw

Load word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.33.1. Encoding



B.33.2. Synopsis

Loads a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1. It expands to `lw rd, offset(rs1)`.

B.33.3. Access

M	S	U
Always	Always	Always

B.33.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> rd = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.33.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<32>(virtual_address, $encoding), 32);
```

B.33.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

B.34. c.lwsp

Load word from stack pointer

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.34.1. Encoding

B.34.2. Synopsis

Loads a 32-bit value from memory into register rd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to lw <code>rd, offset(x2)</code>. C.LWSP is only valid when rd ≠ x0. The code points with rd=x0 are reserved.

B.34.3. Access

M	S	U
Always	Always	Always

B.34.4. Decode Variables

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

B.34.5. Execution

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

B.34.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

B.35. c.mv

Move Register

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.35.1. Encoding

B.35.2. Synopsis

C.MV (move register) performs copy of the data in register rs2 to register rd C.MV expands to addi rd, x0, rs2.

B.35.3. Access

M	S	U
Always	Always	Always

B.35.4. Decode Variables

```
Bits<5> rd = $encoding[11:7];
Bits<5> rs2 = $encoding[6:2];
```

B.35.5. Execution

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

B.35.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

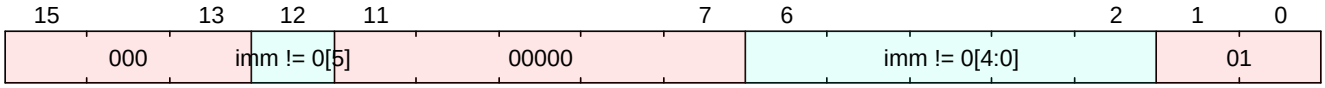
B.36. c.nop

Non-operation

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.36.1. Encoding



B.36.2. Synopsis

C.NOP expands into `addi x0, x0, imm`.

B.36.3. Access

M	S	U
Always	Always	Always

B.36.4. Decode Variables

```
Bits<6> imm = {$encoding[12], $encoding[6:2]};
```

B.36.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
```

B.36.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

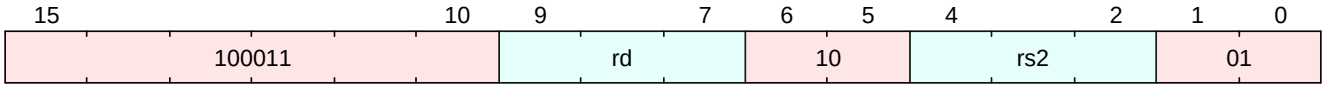
B.37. c.or

Or

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.37.1. Encoding



B.37.2. Synopsis

Or rd with rs2, and store the result in rd The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.OR expands into `or rd, rd, rs2`.

B.37.3. Access

M	S	U
Always	Always	Always

B.37.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.37.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 | t1;
```

B.37.6. Exceptions

This instruction does not generate synchronous exceptions.

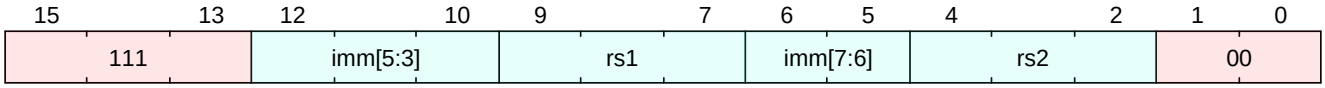
B.38. c.sd

Store double

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.38.1. Encoding



B.38.2. Synopsis

Stores a 64-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the base address in register rs1. It expands to `sd rs2, offset(rs1)`.

B.38.3. Access

M	S	U
Always	Always	Always

B.38.4. Decode Variables

```
Bits<8> imm = {$encoding[6:5], $encoding[12:10], 3'd0};
Bits<3> rs2 = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.38.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
write_memory<64>(virtual_address, X[rs2], $encoding);
```

B.38.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

B.39. c.sdsp

Store doubleword to stack

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.39.1. Encoding

B.39.2. Synopsis

Stores a 64-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 8, to the stack pointer, x2. It expands to `sd rs2, offset(x2)`.

B.39.3. Access

M	S	U
Always	Always	Always

B.39.4. Decode Variables

```
Bits<9> imm = {$encoding[9:7], $encoding[12:10], 3'd0};
Bits<5> rs2 = $encoding[6:2];
```

B.39.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<64>(virtual_address, X[rs2], $encoding);
```

B.39.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

B.40. c.slli

Shift left logical immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.40.1. Encoding

B.40.2. Synopsis

Shift the value in rd left by shamt, and store the result back in rd. C.SLLI expands into `slli rd, rd, shamt`.

B.40.3. Access

M	S	U
Always	Always	Always

B.40.4. Decode Variables

```
Bits<6> shamt = {$encoding[12], $encoding[6:2]};
Bits<5> rd = $encoding[11:7];
```

B.40.5. Execution

```
X[rd] = X[rd] << shamt;
```

B.40.6. Exceptions

This instruction does not generate synchronous exceptions.

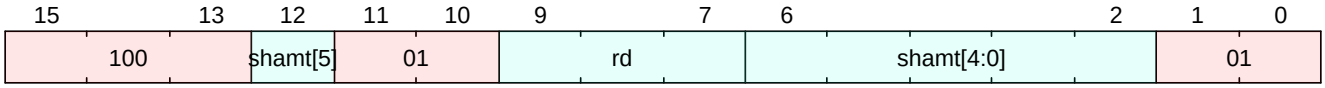
B.41. c.srai

Shift right arithmetical immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.41.1. Encoding



B.41.2. Synopsis

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the value in rd right by shamt, and store the result in rd. The rd register index should be used as rd+8 (registers x8-x15). C.SRAI expands into `srai rd, rd, shamt`.

B.41.3. Access

M	S	U
Always	Always	Always

B.41.4. Decode Variables

```
Bits<6> shamt = {$encoding[12], $encoding[6:2]};
Bits<3> rd = $encoding[9:7];
```

B.41.5. Execution

```
X[rd + 8] = X[rd + 8] >>> shamt;
```

B.41.6. Exceptions

This instruction does not generate synchronous exceptions.

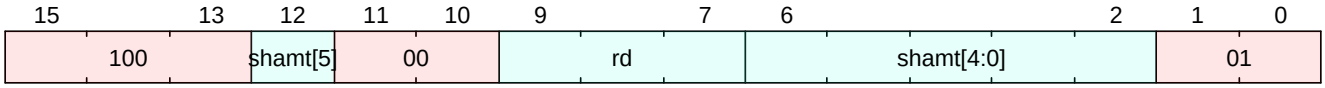
B.42. c.srli

Shift right logical immediate

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.42.1. Encoding



B.42.2. Synopsis

Shift the value in rd right by shamt, and store the result back in rd. The rd register index should be used as rd+8 (registers x8-x15). C.SRLI expands into `srli rd, rd, shamt`.

B.42.3. Access

M	S	U
Always	Always	Always

B.42.4. Decode Variables

```
Bits<6> shamt = {$encoding[12], $encoding[6:2]};
Bits<3> rd = $encoding[9:7];
```

B.42.5. Execution

```
X[rd + 8] = X[rd + 8] >> shamt;
```

B.42.6. Exceptions

This instruction does not generate synchronous exceptions.

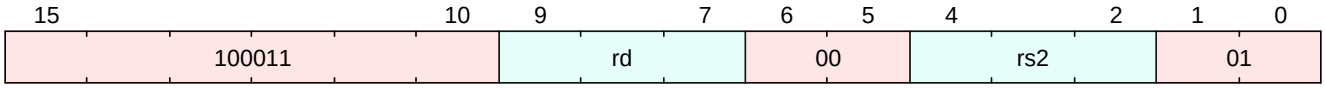
B.43. c.sub

Subtract

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.43.1. Encoding



B.43.2. Synopsis

Subtract the value in rs2 from rd, and store the result in rd. The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.SUB expands into `sub rd, rd, rs2`.

B.43.3. Access

M	S	U
Always	Always	Always

B.43.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.43.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 - t1;
```

B.43.6. Exceptions

This instruction does not generate synchronous exceptions.

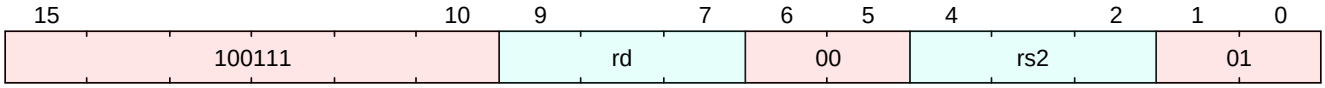
B.44. c.subw

Subtract word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.44.1. Encoding



B.44.2. Synopsis

Subtract the 32-bit values in rs2 from rd, and store the result in rd. The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.SUBW expands into `subw rd, rd, rs2`.

B.44.3. Access

M	S	U
Always	Always	Always

B.44.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.44.5. Execution

```
Bits<32> t0 = X[rd + 8][31:0];
Bits<32> t1 = X[rs2 + 8][31:0];
X[rd + 8] = sext(t0 - t1, 31);
```

B.44.6. Exceptions

This instruction does not generate synchronous exceptions.

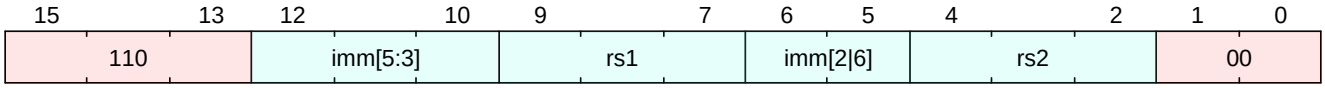
B.45. c.sw

Store word

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.45.1. Encoding



B.45.2. Synopsis

Stores a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register rs1. It expands to `sw rs2, offset(rs1)`.

B.45.3. Access

M	S	U
Always	Always	Always

B.45.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> rs2 = $encoding[4:2];
Bits<3> rs1 = $encoding[9:7];
```

B.45.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1] + imm;
write_memory<32>(virtual_address, X[rs2][31:0], $encoding);
```

B.45.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

B.46. c.swsp

Store word to stack

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.46.1. Encoding

B.46.2. Synopsis

Stores a 32-bit value in register rs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to sw `rs2, offset(x2)`.

B.46.3. Access

M	S	U
Always	Always	Always

B.46.4. Decode Variables

```
Bits<8> imm = {$encoding[8:7], $encoding[12:9], 2'd0};
Bits<5> rs2 = $encoding[6:2];
```

B.46.5. Execution

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<32>(virtual_address, X[rs2][31:0], $encoding);
```

B.46.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault

- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

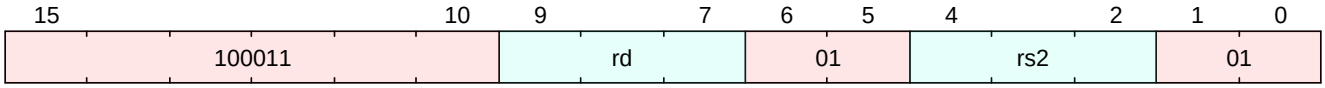
B.47. c.xor

Exclusive Or

This instruction is defined by:

- anyOf:
  - C, version >= 0
  - Zca, version >= 0

B.47.1. Encoding



B.47.2. Synopsis

Exclusive or rd with rs2, and store the result in rd The rd and rs2 register indexes should be used as rd+8 and rs2+8 (registers x8-x15). C.XOR expands into `xor rd, rd, rs2`.

B.47.3. Access

M	S	U
Always	Always	Always

B.47.4. Decode Variables

```
Bits<3> rs2 = $encoding[4:2];
Bits<3> rd = $encoding[9:7];
```

B.47.5. Execution

```
XReg t0 = X[rd + 8];
XReg t1 = X[rs2 + 8];
X[rd + 8] = t0 ^ t1;
```

B.47.6. Exceptions

This instruction does not generate synchronous exceptions.

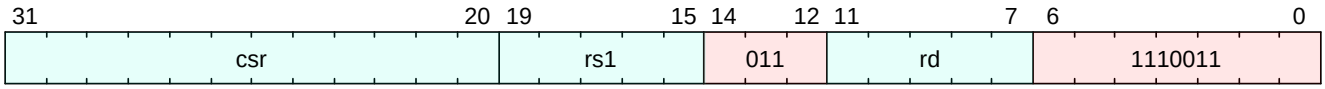
B.48. csrrc

No synopsis available.

This instruction is defined by:

- Zicsr, version >= 0

B.48.1. Encoding



B.48.2. Synopsis

No description available.

B.48.3. Access

M	S	U
Always	Always	Always

B.48.4. Decode Variables

```
Bits<12> csr = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.48.5. Execution

```

```

B.48.6. Exceptions

This instruction does not generate synchronous exceptions.

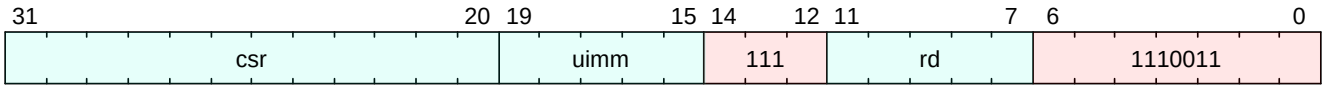
B.49. csrrci

No synopsis available.

This instruction is defined by:

- Zicsr, version >= 0

B.49.1. Encoding



B.49.2. Synopsis

No description available.

B.49.3. Access

M	S	U
Always	Always	Always

B.49.4. Decode Variables

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

B.49.5. Execution

```

```

B.49.6. Exceptions

This instruction does not generate synchronous exceptions.

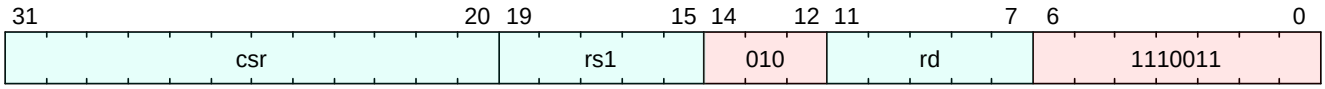
B.50. csrrs

Atomic Read and Set Bits in CSR

This instruction is defined by:

- Zicsr, version >= 0

B.50.1. Encoding



B.50.2. Synopsis

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 `rd`. The initial value in integer register `rs1` is treated as a bit mask that specifies bit positions to be set in the CSR. Any bit that is high in `rs1` 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.50.3. Access

M	S	U
Always	Always	Always

B.50.4. Decode Variables

```
Bits<12> csr = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.50.5. Execution

```
XReg initial_csr_value = CSR[csr].sw_read();
XReg mask = X[rs1];
CSR[csr].sw_write(initial_csr_value | mask);
X[rd] = initial_csr_value;
```

B.50.6. Exceptions

This instruction does not generate synchronous exceptions.

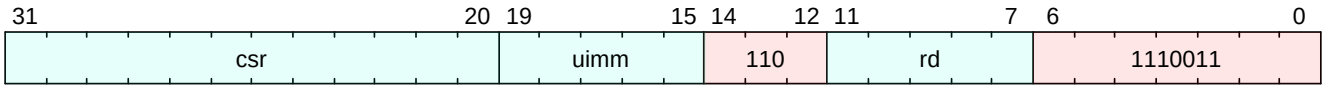
B.51. csrrsi

No synopsis available.

This instruction is defined by:

- Zicsr, version >= 0

B.51.1. Encoding



B.51.2. Synopsis

No description available.

B.51.3. Access

M	S	U
Always	Always	Always

B.51.4. Decode Variables

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

B.51.5. Execution

```

```

B.51.6. Exceptions

This instruction does not generate synchronous exceptions.

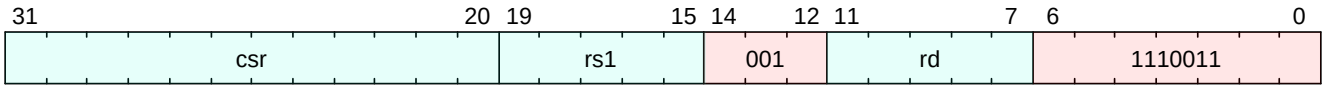
B.52. csrrw

Atomic Read/Write CSR

This instruction is defined by:

- Zicsr, version >= 0

B.52.1. Encoding



B.52.2. Synopsis

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 rd. The initial value in rs1 is written to the CSR. If `rd=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.52.3. Access

M	S	U
Always	Always	Always

B.52.4. Decode Variables

```
Bits<12> csr = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.52.5. Execution

```
if (rd != 0) {
  X[rd] = CSR[csr].sw_read();
}
CSR[csr].sw_write(X[rs1]);
```

B.52.6. Exceptions

This instruction does not generate synchronous exceptions.

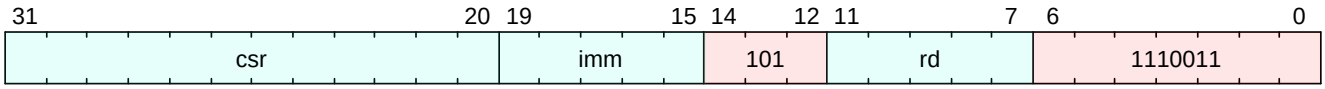
B.53. csrrwi

Atomic Read/Write CSR Immediate

This instruction is defined by:

- Zicsr, version >= 0

B.53.1. Encoding



B.53.2. Synopsis

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

Read the old value of the CSR, zero-extends the value to `XLEN` bits, and then write it to integer register rd. The 5-bit uimm field is zero-extended and written to the CSR. If `rd=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.53.3. Access

M	S	U
Always	Always	Always

B.53.4. Decode Variables

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

B.53.5. Execution

```
if (rd != 0) {
  X[rd] = CSR[csr].sw_read();
}
CSR[csr].sw_write({{XLEN - 5{1'b0}}, imm});
```

B.53.6. Exceptions

This instruction does not generate synchronous exceptions.

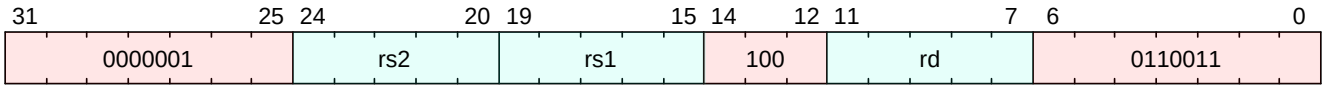
B.54. div

Signed division

This instruction is defined by:

- M, version >= 0

B.54.1. Encoding



B.54.2. Synopsis

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.54.3. Access

M	S	U
Always	Always	Always

B.54.4. Decode Variables

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

B.54.5. Execution

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

```
}
```

B.54.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

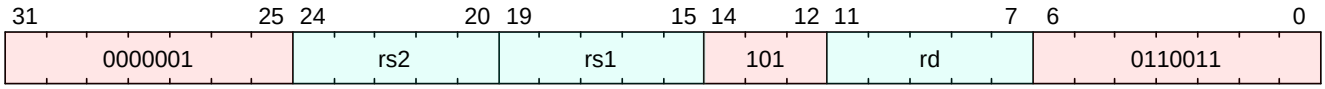
B.55. divu

Unsigned division

This instruction is defined by:

- M, version >= 0

B.55.1. Encoding



B.55.2. Synopsis

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.55.3. Access

M	S	U
Always	Always	Always

B.55.4. Decode Variables

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

B.55.5. Execution

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

B.55.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

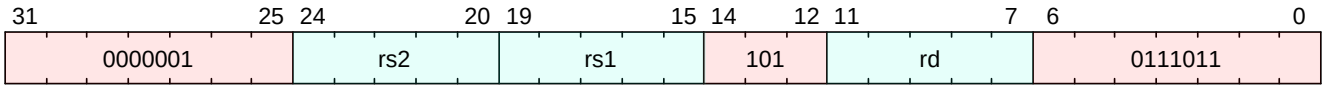
B.56. divuw

Unsigned 32-bit division

This instruction is defined by:

- M, version >= 0

B.56.1. Encoding



B.56.2. Synopsis

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.56.3. Access

M	S	U
Always	Always	Always

B.56.4. Decode Variables

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

B.56.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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.56.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

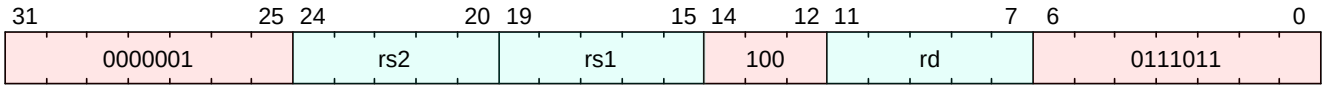
B.57. divw

Signed 32-bit division

This instruction is defined by:

- M, version >= 0

B.57.1. Encoding



B.57.2. Synopsis

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.57.3. Access

M	S	U
Always	Always	Always

B.57.4. Decode Variables

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

B.57.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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] = {XLEN{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.57.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

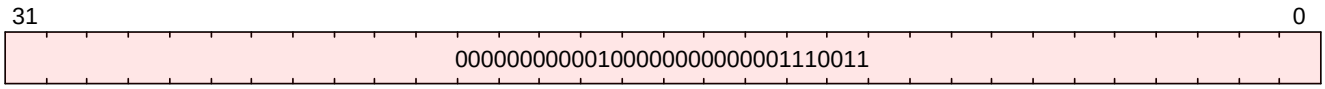
B.58. ebreak

Breakpoint exception

This instruction is defined by:

- I, version >= 0

B.58.1. Encoding



B.58.2. Synopsis

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.

NOTE	As described in the C Standard 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.58.3. Access

M	S	U
Always	Always	Always

B.58.4. Decode Variables

```
```

B.58.5. Execution

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

B.58.6. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

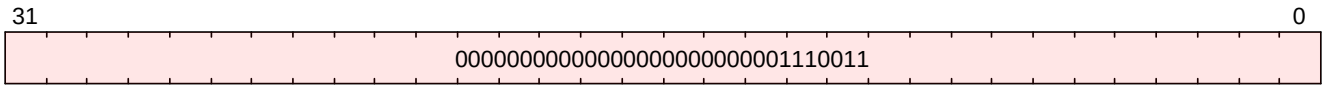
B.59. ecall

Environment call

This instruction is defined by:

- I, version >= 0

B.59.1. Encoding



B.59.2. Synopsis

The ECALL instruction is used to make 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.

NOTE	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.59.3. Access

M	S	U
Always	Always	Always

B.59.4. Decode Variables

```
```

B.59.5. Execution

```
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.59.6. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

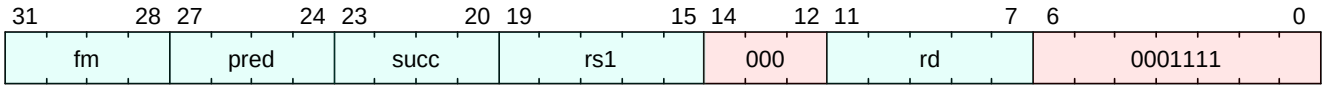
B.60. fence

Memory ordering fence

This instruction is defined by:

- I, version >= 0

B.60.1. Encoding



B.60.2. Synopsis

Orders memory operations.

The fence instruction is used to order device I/O and memory accesses as viewed by other RISC-V harts and external devices or coprocessors. Any combination of device input (I), device output (O), memory reads (R), and memory writes (W) may be ordered with respect to any combination of the same. Informally, no other RISC-V hart or external device can observe any operation in the *successor* set following a fence before any operation in the *predecessor* set preceding the fence.

The predecessor and successor fields have the same format to specify operation types:

`pred`				`succ`			
27	26	25	24	23	22	21	20
PI	PO	PR	PW	SI	SO	SR	SW

Table 6. Fence mode encoding

fm field	Mnemonic	Meaning
0000	*none*	Normal Fence
1000	TSO	With `FENCE RW,RW`: exclude write-to-read ordering; otherwise: *Reserved for future use.*
other		*Reserved for future use.*

When the mode field *fm* is `0001` and both the predecessor and successor sets are 'RW', then the instruction acts as a special-case `fence.tso`. `fence.tso` orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the 'fence.tso's predecessor set unordered with non-AMO loads in its successor set.

When mode field *fm* is not `0001`, or when mode field *fm* is `0001` but the *pred* and *succ* fields are not both 'RW' (0x3), then the fence acts as a baseline fence (*e.g.*, *fm* is effectively `0000`). This is unaffected by the FIOM bits, described below (implicit promotion does not change how `fence.tso` is decoded).

The rs1 and rd fields are unused and ignored.

In modes other than M-mode, fence is further affected by menvcfg.FIOM, senvcfg.FIOM<% if ext?(:H) %>, and/or henvcfg.FIOM<% end %> as follows:

Table 7. Effective PR/PW/SR/SW in (H)S-mode

menvcfg.FIOM	pred.PI pred.PO succ.SI succ.SO	→ → → →	**effective** PR **effective** PW **effective** SR **effective** SW
0	-		from encoding
1	0		from encoding
1	1		1

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

menvcfg.FIOM	**senvcfg.FIOM**	pred.PI pred.PO succ.SI succ.SO	→ → → →	**effective** PR **effective** PW **effective** SR **effective** SW
0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding
1	-	1		1

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

menvcfg.FIOM	**henvcfg.FIOM**	pred.PI pred.PO succ.SI succ.SO	→ → → →	**effective** PR **effective** PW **effective** SR **effective** SW
0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding
1	-	1		1

<%- end -%>

B.60.3. Access

M	S	U

Always	Always	Always

B.60.4. Decode Variables

```
Bits<4> fm = $encoding[31:28];
Bits<4> pred = $encoding[27:24];
Bits<4> succ = $encoding[23:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.60.5. Execution

```
Boolean is_fence_tso;
Boolean is_pause;
if (fm == 1) {
  if (pred == 0x3 && succ == 0x3) {
    is_fence_tso = true;
  }
}
if (implemented?(ExtensionName::Zihintpause)) {
  if ((pred == 1) && (succ == 0) && (fm == 0) && (rd == 0) && (rs1 == 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_fence_tso) {
  fence_tso();
} else if (is_pause) {
  pause();
} else {
  if (mode() == PrivilegeMode::S) {
    if (CSR[menvcfg].FIOM == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
```

```
      if (succ_o) {
        succ_w = true;
      }
    }
  } else if (mode() == PrivilegeMode::U) {
    if ((CSR[menvcfg].FIOM | CSR[senvcfg].FIOM) == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  } else if (mode() == PrivilegeMode::VS || mode() == PrivilegeMode::VU) {
    if ((CSR[menvcfg].FIOM | CSR[henvcfg].FIOM) == 1) {
      if (pred_i) {
        pred_r = true;
      }
      if (pred_o) {
        pred_w = true;
      }
      if (succ_i) {
        succ_r = true;
      }
      if (succ_o) {
        succ_w = true;
      }
    }
  }
  fence(pred_i, pred_o, pred_r, pred_w, succ_i, succ_o, succ_r, succ_w);
}
```

B.60.6. Exceptions

This instruction does not generate synchronous exceptions.

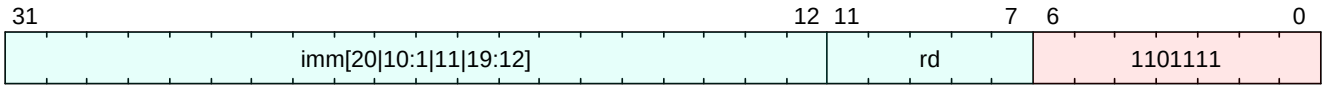
B.61. jal

Jump and link

This instruction is defined by:

- I, version >= 0

B.61.1. Encoding



B.61.2. Synopsis

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

B.61.3. Access

M	S	U
Always	Always	Always

B.61.4. Decode Variables

```
signed Bits<21> imm = sext({$encoding[31], $encoding[19:12], $encoding[20],
$encoding[30:21], 1'd0});
Bits<5> rd = $encoding[11:7];
```

B.61.5. Execution

```
XReg retrun_addr = $pc + 4;
jump_halfword($pc + imm);
X[rd] = retrun_addr;
```

B.61.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

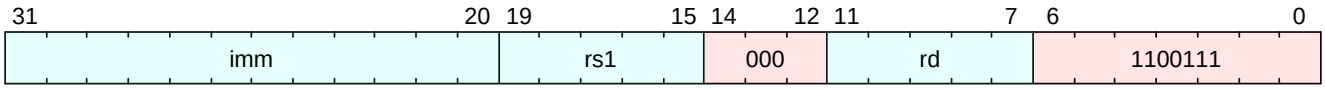
B.62. jalr

Jump and link register

This instruction is defined by:

- I, version >= 0

B.62.1. Encoding



B.62.2. Synopsis

Jump to an address formed by adding rs1 to a signed offset then clearing the least significant bit, and store the return address in rd.

B.62.3. Access

M	S	U
Always	Always	Always

B.62.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.62.5. Execution

```
XReg returnaddr;
returnaddr = $pc + 4;
jump((X[rs1] + imm) & ~XLEN'1);
X[rd] = returnaddr;
```

B.62.6. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

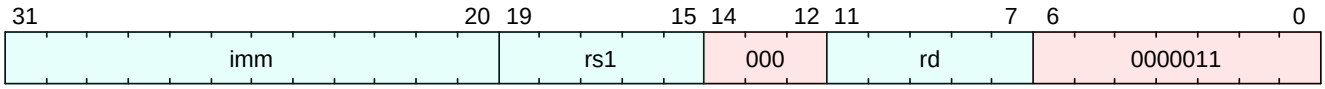
B.63. lb

Load byte

This instruction is defined by:

- I, version >= 0

B.63.1. Encoding



B.63.2. Synopsis

Load 8 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset. Sign extend the result.

B.63.3. Access

M	S	U
Always	Always	Always

B.63.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.63.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<8>(virtual_address, $encoding), 8);
```

B.63.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

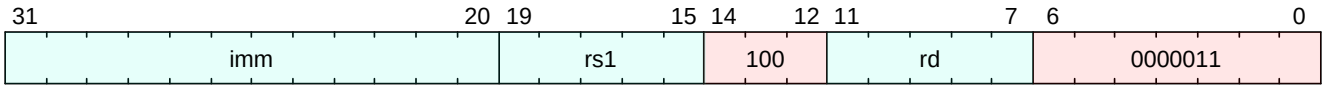
B.64. lbu

Load byte unsigned

This instruction is defined by:

- I, version >= 0

B.64.1. Encoding



B.64.2. Synopsis

Load 8 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset. Zero extend the result.

B.64.3. Access

M	S	U
Always	Always	Always

B.64.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.64.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<8>(virtual_address, $encoding);
```

B.64.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

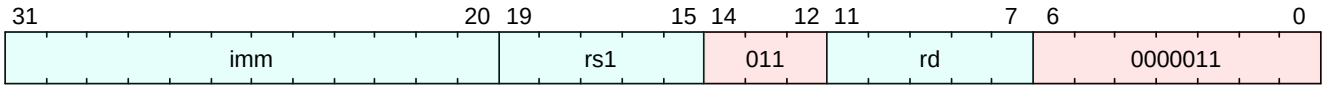
B.65. ld

Load doubleword

This instruction is defined by:

- I, version >= 0

B.65.1. Encoding



B.65.2. Synopsis

Load 64 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset.

B.65.3. Access

M	S	U
Always	Always	Always

B.65.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.65.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<64>(virtual_address, $encoding);
```

B.65.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

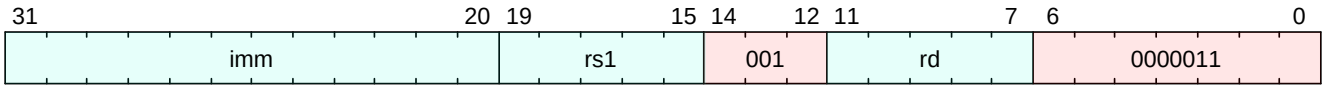
B.66. lh

Load halfword

This instruction is defined by:

- I, version >= 0

B.66.1. Encoding



B.66.2. Synopsis

Load 16 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset. Sign extend the result.

B.66.3. Access

M	S	U
Always	Always	Always

B.66.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.66.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = sext(read_memory<16>(virtual_address, $encoding), 16);
```

B.66.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

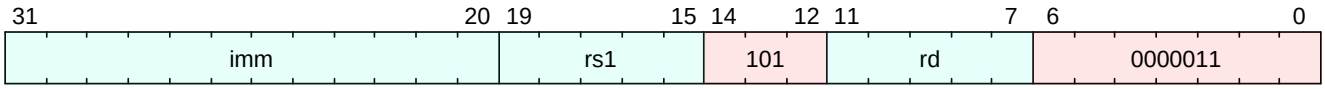
B.67. lhu

Load halfword unsigned

This instruction is defined by:

- I, version >= 0

B.67.1. Encoding



B.67.2. Synopsis

Load 16 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset. Zero extend the result.

B.67.3. Access

M	S	U
Always	Always	Always

B.67.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.67.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<16>(virtual_address, $encoding);
```

B.67.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

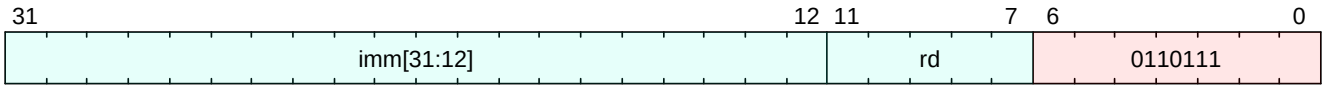
B.68. lui

Load upper immediate

This instruction is defined by:

- I, version >= 0

B.68.1. Encoding



B.68.2. Synopsis

Load the zero-extended imm into rd.

B.68.3. Access

M	S	U
Always	Always	Always

B.68.4. Decode Variables

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

B.68.5. Execution

```
X[rd] = imm;
```

B.68.6. Exceptions

This instruction does not generate synchronous exceptions.

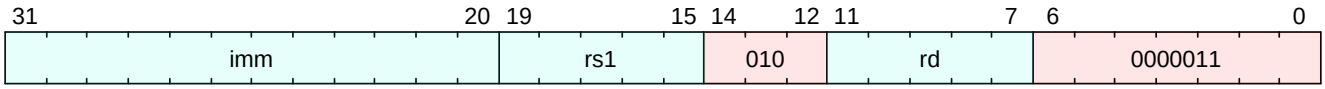
B.69. lw

Load word

This instruction is defined by:

- I, version >= 0

B.69.1. Encoding



B.69.2. Synopsis

Load 32 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset. Sign extend the result.

B.69.3. Access

M	S	U
Always	Always	Always

B.69.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.69.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<32>(virtual_address, $encoding);
```

B.69.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

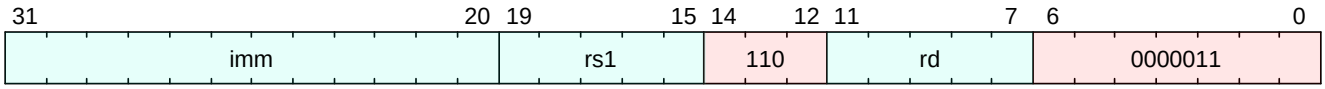
B.70. lwu

Load word unsigned

This instruction is defined by:

- I, version >= 0

B.70.1. Encoding



B.70.2. Synopsis

Load 64 bits of data into register `rd` from an address formed by adding `rs1` to a signed offset. Zero extend the result.

B.70.3. Access

M	S	U
Always	Always	Always

B.70.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.70.5. Execution

```
XReg virtual_address = X[rs1] + imm;
X[rd] = read_memory<32>(virtual_address, $encoding);
```

B.70.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

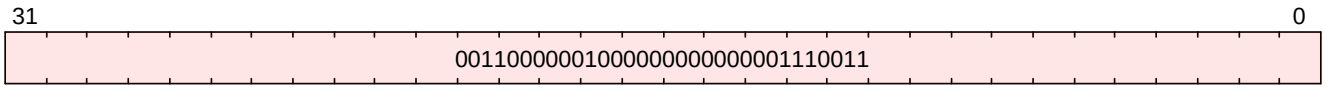
B.71. mret

Machine Exception Return

This instruction is defined by:

- Sm, version >= 0

B.71.1. Encoding



B.71.2. Synopsis

Returns from an exception in M-mode.

B.71.3. Access

M	S	U
Always	Never	Never

B.71.4. Decode Variables

```
```

B.71.5. Execution

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

B.71.6. Exceptions

This instruction does not generate synchronous exceptions.

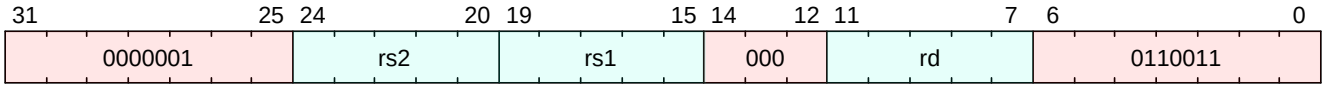
B.72. mul

Signed multiply

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.72.1. Encoding



B.72.2. Synopsis

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.

NOTE 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.72.3. Access

M	S	U
Always	Always	Always

B.72.4. Decode Variables

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

B.72.5. Execution

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

```
X[rd] = (src1 * src2)[XLEN - 1:0];
```

B.72.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

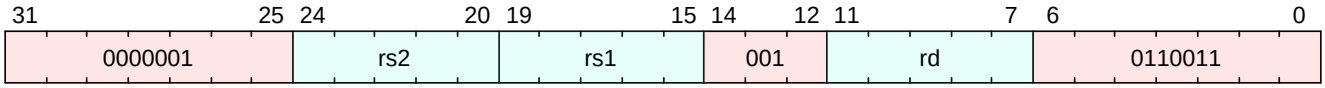
B.73. mulh

Signed multiply high

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.73.1. Encoding



B.73.2. Synopsis

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.73.3. Access

M	S	U
Always	Always	Always

B.73.4. Decode Variables

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

B.73.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
}
Bits<1> rs1_sign_bit = X[rs1][xlen() - 1];
Bits<XLEN * 2> src1 = {{xlen(){rs1_sign_bit}}, X[rs1]};
Bits<1> rs2_sign_bit = X[rs2][xlen() - 1];
Bits<XLEN * 2> src2 = {{xlen(){rs2_sign_bit}}, X[rs2]};
X[rd] = (src1 * src2)[(xlen() * 8'd2) - 1:xlen()];
```

B.73.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

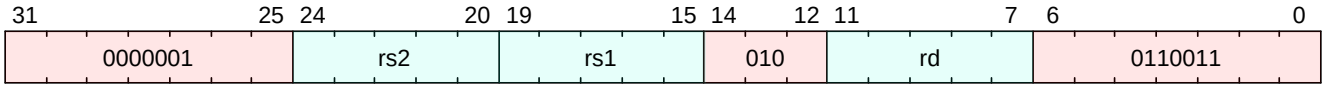
B.74. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.74.1. Encoding



B.74.2. Synopsis

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.74.3. Access

M	S	U
Always	Always	Always

B.74.4. Decode Variables

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

B.74.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
}
Bits<1> rs1_sign_bit = X[rs1][XLEN - 1];
Bits<XLEN * 8'd2> src1 = {{XLEN{rs1_sign_bit}}, X[rs1]};
Bits<XLEN * 8'd2> src2 = {{XLEN{1'b0}}, X[rs2]};
X[rd] = (src1 * src2)[(XLEN * 8'd2) - 1:XLEN];
```

B.74.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

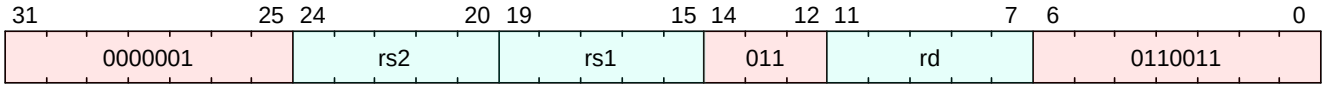
B.75. mulhu

Unsigned multiply high

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.75.1. Encoding



B.75.2. Synopsis

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.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. Execution

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
```

```
}
Bits<XLEN * 8'd2> src1 = {{XLEN{1'b0}}, X[rs1]};
Bits<XLEN * 8'd2> src2 = {{XLEN{1'b0}}, X[rs2]};
X[rd] = (src1 * src2)[(XLEN * 8'd2) - 1:XLEN];
```

B.75.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

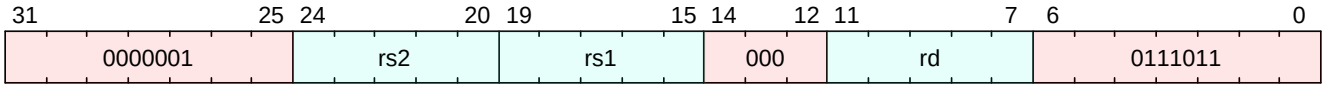
B.76. mulw

Signed 32-bit multiply

This instruction is defined by:

- anyOf:
  - M, version >= 0
  - Zmmul, version >= 0

B.76.1. Encoding



B.76.2. Synopsis

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.

NOTE	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.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. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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.76.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

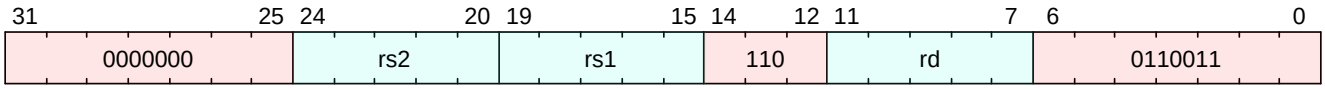
B.77. or

Or

This instruction is defined by:

- I, version >= 0

B.77.1. Encoding



B.77.2. Synopsis

Or rs1 with 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. Execution

```
X[rd] = X[rs1] | X[rs2];
```

B.77.6. Exceptions

This instruction does not generate synchronous exceptions.

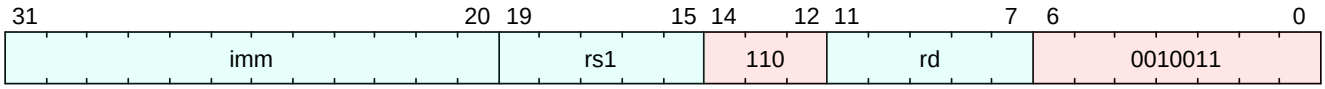
B.78. ori

Or immediate

This instruction is defined by:

- I, version >= 0

B.78.1. Encoding



B.78.2. Synopsis

Or an immediate to the value in rs1, and store the result in rd

B.78.3. Access

M	S	U
Always	Always	Always

B.78.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.78.5. Execution

```
if (implemented?(ExtensionName::Zicbop)) {
  if (rd == 0) {
    if (imm[4:0] == 0) {
      Bits<12> offset = {imm[11:5], rd};
      prefetch_instruction(offset);
    } else if (imm[4:0] == 1) {
      Bits<12> offset = {imm[11:5], rd};
      prefetch_read(offset);
    } else if (imm[4:0] == 3) {
      Bits<12> offset = {imm[11:5], rd};
      prefetch_write(offset);
    }
  }
}
X[rd] = X[rs1] | imm;
```

B.78.6. Exceptions

This instruction does not generate synchronous exceptions.

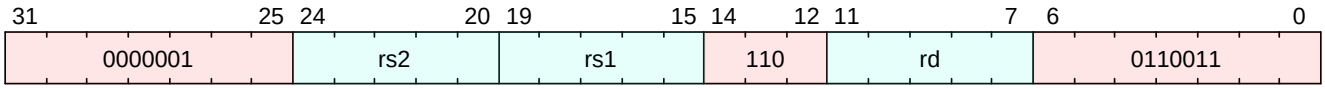
B.79. rem

Signed remainder

This instruction is defined by:

- M, version >= 0

B.79.1. Encoding



B.79.2. Synopsis

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.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. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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, {XLEN - 1{1'b0}}}) && (src2 == {XLEN{1'b1}})) {
  X[rd] = 0;
} else {
  X[rd] = $signed(src1) % $signed(src2);
}
```

B.79.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

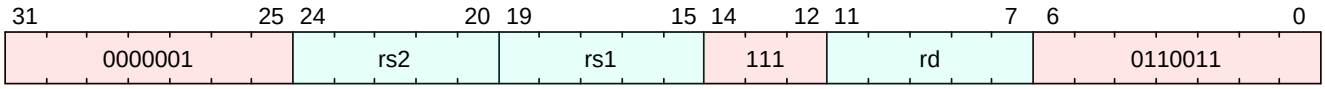
B.80. remu

Unsigned remainder

This instruction is defined by:

- M, version >= 0

B.80.1. Encoding



B.80.2. Synopsis

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

B.80.3. Access

M	S	U
Always	Always	Always

B.80.4. Decode Variables

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

B.80.5. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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.80.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

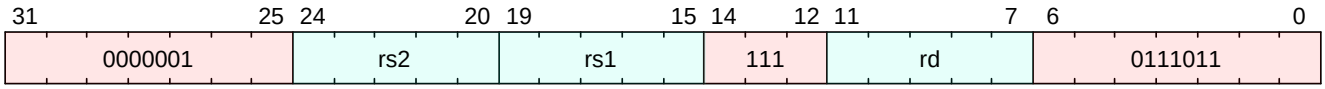
B.81. remuw

Unsigned 32-bit remainder

This instruction is defined by:

- M, version >= 0

B.81.1. Encoding



B.81.2. Synopsis

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.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. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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.81.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

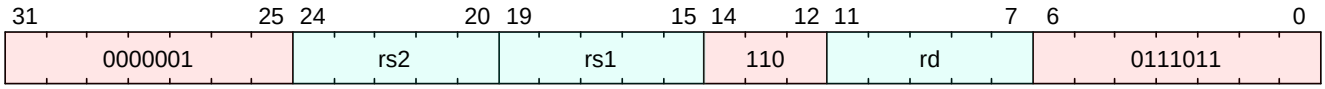
B.82. remw

Signed 32-bit remainder

This instruction is defined by:

- M, version >= 0

B.82.1. Encoding



B.82.2. Synopsis

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.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. Execution

```
if (implemented?(ExtensionName::M) && (CSR[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.82.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

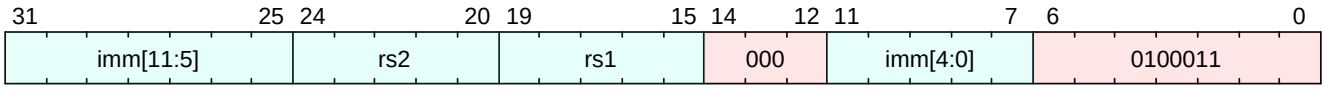
B.83. sb

Store byte

This instruction is defined by:

- I, version >= 0

B.83.1. Encoding



B.83.2. Synopsis

Store 8 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.83.3. Access

M	S	U
Always	Always	Always

B.83.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.83.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<8>(virtual_address, X[rs2][7:0], $encoding);
```

B.83.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

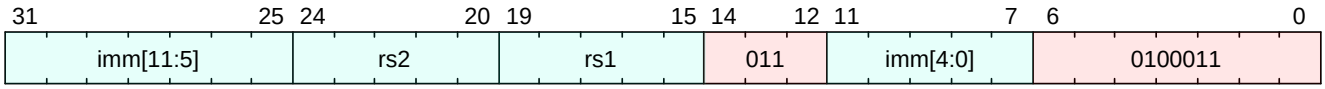
B.84. sd

Store doubleword

This instruction is defined by:

- I, version >= 0

B.84.1. Encoding



B.84.2. Synopsis

Store 64 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.84.3. Access

M	S	U
Always	Always	Always

B.84.4. Decode Variables

```
signed Bits<12> imm = sext({$encoding[31:25], $encoding[11:7]});
Bits<5> rs1 = $encoding[19:15];
Bits<5> rs2 = $encoding[24:20];
```

B.84.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<64>(virtual_address, X[rs2], $encoding);
```

B.84.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

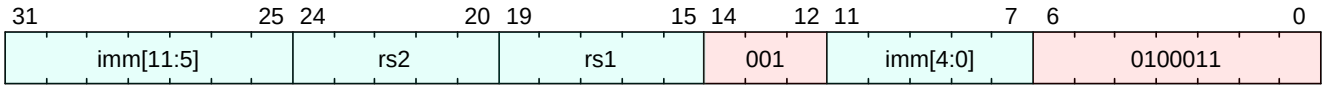
B.85. sh

Store halfword

This instruction is defined by:

- I, version >= 0

B.85.1. Encoding



B.85.2. Synopsis

Store 16 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.85.3. Access

M	S	U
Always	Always	Always

B.85.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.85.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<16>(virtual_address, X[rs2][15:0], $encoding);
```

B.85.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

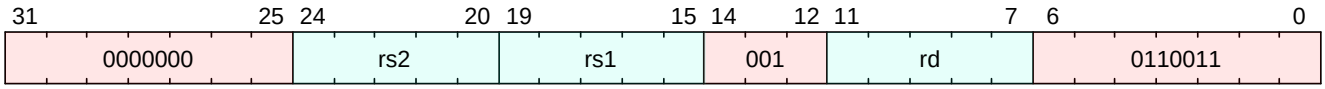
B.86. sll

Shift left logical

This instruction is defined by:

- I, version >= 0

B.86.1. Encoding



B.86.2. Synopsis

Shift the value in `rs1` left by the value in the lower 6 bits of `rs2`, and store the result in `rd`.

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. Execution

```
if (xlen() == 64) {
  X[rd] = X[rs1] << X[rs2][5:0];
} else {
  X[rd] = X[rs1] << X[rs2][4:0];
}
```

B.86.6. Exceptions

This instruction does not generate synchronous exceptions.

B.87. slli

Shift left logical immediate

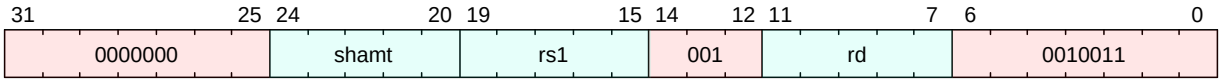
This instruction is defined by:

- I, version >= 0

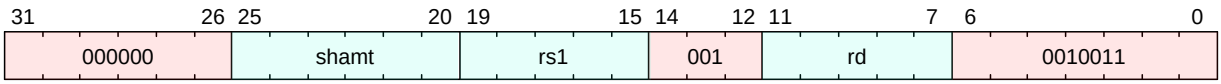
B.87.1. Encoding

NOTE This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.87.2. Synopsis

Shift the value in rs1 left by shamt, and store the result in rd

B.87.3. Access

M	S	U
Always	Always	Always

B.87.4. 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.87.5. Execution

```
X[rd] = X[rs1] << shamt;
```

B.87.6. Exceptions

This instruction does not generate synchronous exceptions.

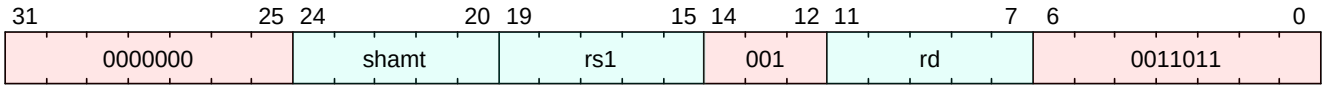
B.88. slliw

Shift left logical immediate word

This instruction is defined by:

- I, version >= 0

B.88.1. Encoding



B.88.2. Synopsis

Shift the 32-bit value in rs1 left by shamt, and store the sign-extended result in rd

B.88.3. Access

M	S	U
Always	Always	Always

B.88.4. Decode Variables

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

B.88.5. Execution

```
X[rd] = sext(X[rs1] << shamt, 31);
```

B.88.6. Exceptions

This instruction does not generate synchronous exceptions.

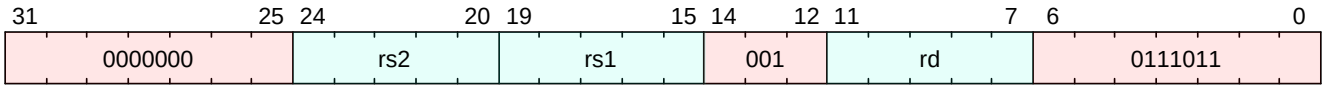
B.89. sllw

Shift left logical word

This instruction is defined by:

- I, version >= 0

B.89.1. Encoding



B.89.2. Synopsis

Shift the 32-bit value in `rs1` left by the value in the lower 5 bits of `rs2`, and store the sign-extended result in `rd`.

B.89.3. Access

M	S	U
Always	Always	Always

B.89.4. Decode Variables

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

B.89.5. Execution

```
X[rd] = sext(X[rs1] << X[rs2][4:0], 31);
```

B.89.6. Exceptions

This instruction does not generate synchronous exceptions.

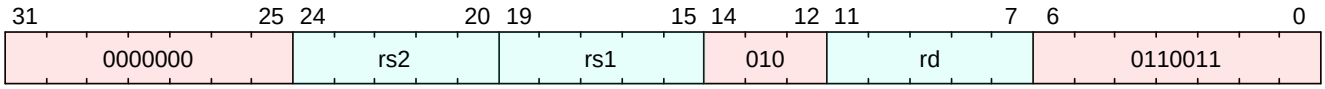
B.90. slt

Set on less than

This instruction is defined by:

- I, version >= 0

B.90.1. Encoding



B.90.2. Synopsis

Places the value 1 in register `rd` if register `rs1` is less than the value in register `rs2`, where both sources are treated as signed numbers, else 0 is written to `rd`.

B.90.3. Access

M	S	U
Always	Always	Always

B.90.4. Decode Variables

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

B.90.5. Execution

```
XReg src1 = X[rs1];
XReg src2 = X[rs2];
X[rd] = ($signed(src1) < $signed(src2)) ? '1 : '0;
```

B.90.6. Exceptions

This instruction does not generate synchronous exceptions.

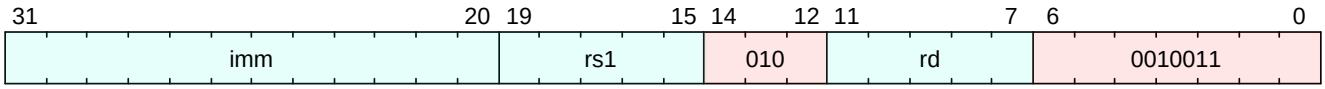
B.91. slti

Set on less than immediate

This instruction is defined by:

- I, version >= 0

B.91.1. Encoding



B.91.2. Synopsis

Places the value 1 in register `rd` if register `rs1` is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to `rd`.

B.91.3. Access

M	S	U
Always	Always	Always

B.91.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.91.5. Execution

```
X[rd] = ($signed(X[rs1]) < $signed(imm)) ? '1 : '0;
```

B.91.6. Exceptions

This instruction does not generate synchronous exceptions.

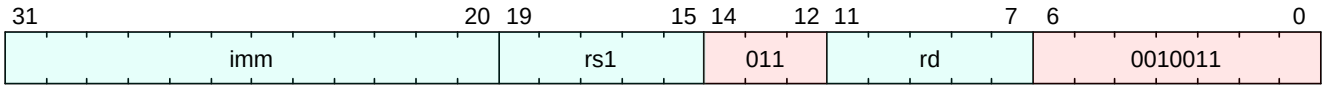
B.92. sltiu

Set on less than immediate unsigned

This instruction is defined by:

- I, version >= 0

B.92.1. Encoding



B.92.2. Synopsis

Places the value 1 in register `rd` if register `rs1` 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 `rd`.

NOTE `sltiu rd, rs1, 1` sets `rd` to 1 if `rs1` equals zero, otherwise sets `rd` to 0 (assembler pseudoinstruction `SEQZ rd, rs`).

B.92.3. Access

M	S	U
Always	Always	Always

B.92.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.92.5. Execution

```
X[rd] = (X[rs1] < imm) ? 1 : 0;
```

B.92.6. Exceptions

This instruction does not generate synchronous exceptions.

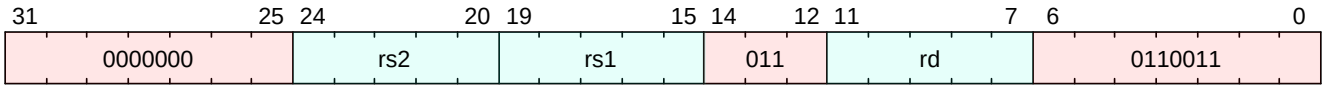
B.93. sltu

Set on less than unsigned

This instruction is defined by:

- I, version >= 0

B.93.1. Encoding



B.93.2. Synopsis

Places the value 1 in register `rd` if register `rs1` is less than the value in register `rs2`, where both sources are treated as unsigned numbers, else 0 is written to `rd`.

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. Execution

```
X[rd] = (X[rs1] < X[rs2]) ? 1 : 0;
```

B.93.6. Exceptions

This instruction does not generate synchronous exceptions.

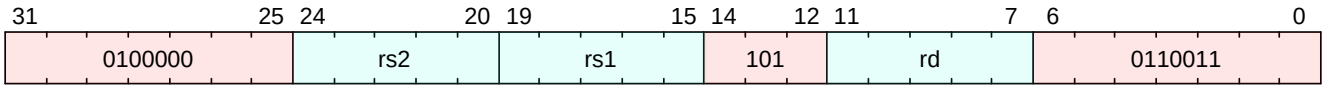
B.94. sra

Shift right arithmetic

This instruction is defined by:

- I, version >= 0

B.94.1. Encoding



B.94.2. Synopsis

Arithmetic shift the value in `rs1` right by the value in the lower 5 bits of `rs2`, and store the result in `rd`.

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. Execution

```
if (xlen() == 64) {
  X[rd] = X[rs1] >>> X[rs2][5:0];
} else {
  X[rd] = X[rs1] >>> X[rs2][4:0];
}
```

B.94.6. Exceptions

This instruction does not generate synchronous exceptions.

B.95. srai

Shift right arithmetic immediate

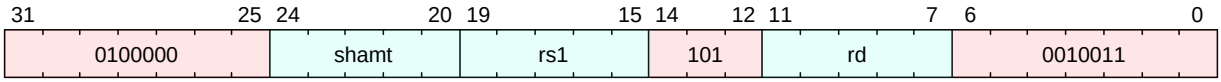
This instruction is defined by:

- I, version >= 0

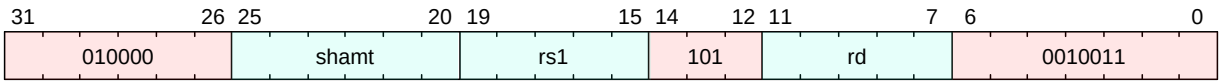
B.95.1. Encoding

NOTE	This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.95.2. Synopsis

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the value in rs1 right by shamt, and store the result in rd.

B.95.3. Access

M	S	U
Always	Always	Always

B.95.4. 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.95.5. Execution

```
X[rd] = X[rs1] >>> shamt;
```

B.95.6. Exceptions

This instruction does not generate synchronous exceptions.

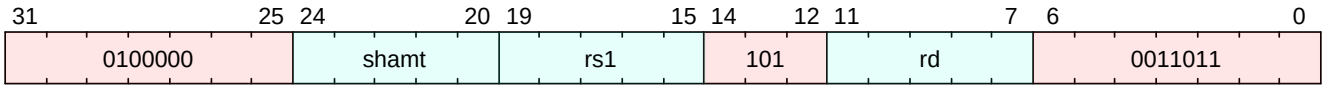
B.96. sraiw

Shift right arithmetic immediate word

This instruction is defined by:

- I, version >= 0

B.96.1. Encoding



B.96.2. Synopsis

Arithmetic shift (the original sign bit is copied into the vacated upper bits) the 32-bit value in rs1 right by shamt, and store the sign-extended result in rd.

B.96.3. Access

M	S	U
Always	Always	Always

B.96.4. Decode Variables

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

B.96.5. Execution

```
XReg operand = sext(X[rs1], 31);
X[rd] = sext(operand >>> shamt, 31);
```

B.96.6. Exceptions

This instruction does not generate synchronous exceptions.

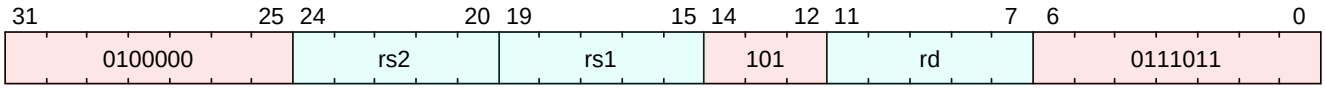
B.97. sraw

Shift right arithmetic word

This instruction is defined by:

- I, version >= 0

B.97.1. Encoding



B.97.2. Synopsis

Arithmetic shift the 32-bit value in `rs1` right by the value in the lower 5 bits of `rs2`, and store the sign-extended result in `rd`.

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. Execution

```
XReg operand1 = sext(X[rs1], 31);
X[rd] = sext(operand1 >>> X[rs2][4:0], 31);
```

B.97.6. Exceptions

This instruction does not generate synchronous exceptions.

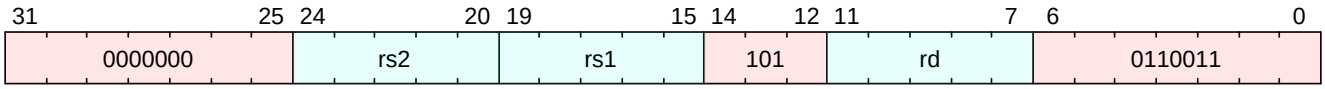
B.98. srl

Shift right logical

This instruction is defined by:

- I, version >= 0

B.98.1. Encoding



B.98.2. Synopsis

Logical shift the value in `rs1` right by the value in the lower bits of `rs2`, and store the result in `rd`.

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. Execution

```
if (xlen() == 64) {
  X[rd] = X[rs1] >> X[rs2][5:0];
} else {
  X[rd] = X[rs1] >> X[rs2][4:0];
}
```

B.98.6. Exceptions

This instruction does not generate synchronous exceptions.

B.99. srli

Shift right logical immediate

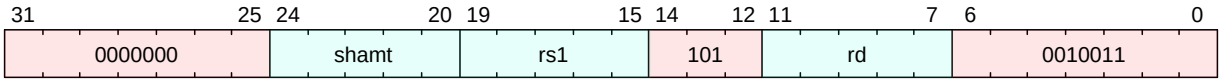
This instruction is defined by:

- I, version >= 0

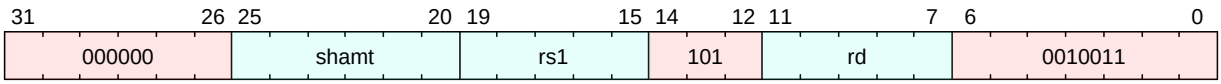
B.99.1. Encoding

NOTE This instruction has different encodings in RV32 and RV64.

RV32



RV64



B.99.2. Synopsis

Shift the value in rs1 right by shamt, and store the result in rd

B.99.3. Access

M	S	U
Always	Always	Always

B.99.4. 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.99.5. Execution

```
X[rd] = X[rs1] >> shamt;
```

B.99.6. Exceptions

This instruction does not generate synchronous exceptions.

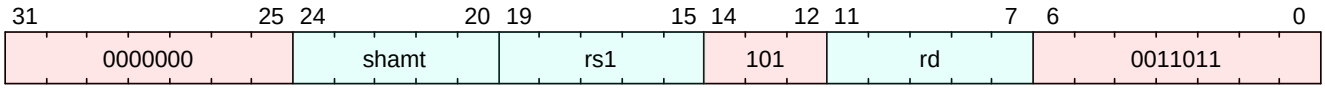
B.100. srliw

Shift right logical immediate word

This instruction is defined by:

- I, version >= 0

B.100.1. Encoding



B.100.2. Synopsis

Shift the 32-bit value in rs1 right by shamt, and store the sign-extended result in rd

B.100.3. Access

M	S	U
Always	Always	Always

B.100.4. Decode Variables

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

B.100.5. Execution

```
XReg operand = X[rs1][31:0];
X[rd] = sext(operand >> shamt, 31);
```

B.100.6. Exceptions

This instruction does not generate synchronous exceptions.

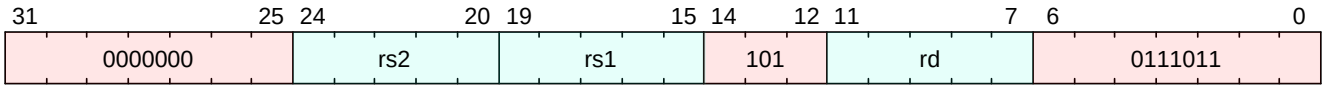
B.101. srlw

Shift right logical word

This instruction is defined by:

- I, version >= 0

B.101.1. Encoding



B.101.2. Synopsis

Logical shift the 32-bit value in `rs1` right by the value in the lower 5 bits of `rs2`, and store the sign-extended result in `rd`.

B.101.3. Access

M	S	U
Always	Always	Always

B.101.4. Decode Variables

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

B.101.5. Execution

```
X[rd] = sext(X[rs1][31:0] >> X[rs2][4:0], 31);
```

B.101.6. Exceptions

This instruction does not generate synchronous exceptions.

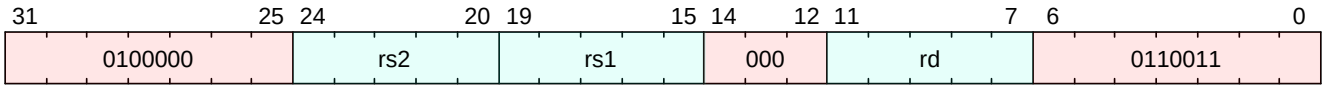
B.102. sub

Subtract

This instruction is defined by:

- I, version >= 0

B.102.1. Encoding



B.102.2. Synopsis

Subtract the value in rs2 from rs1, and store the result in rd

B.102.3. Access

M	S	U
Always	Always	Always

B.102.4. Decode Variables

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

B.102.5. Execution

```
XReg t0 = X[rs1];
XReg t1 = X[rs2];
X[rd] = t0 - t1;
```

B.102.6. Exceptions

This instruction does not generate synchronous exceptions.

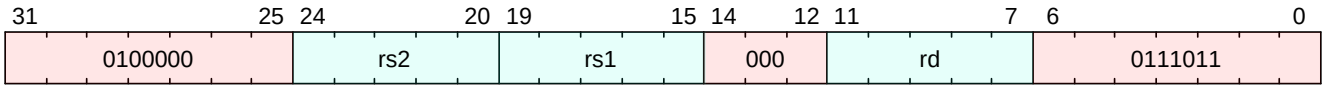
B.103. subw

Subtract word

This instruction is defined by:

- I, version >= 0

B.103.1. Encoding



B.103.2. Synopsis

Subtract the 32-bit values in rs2 from rs1, and store the sign-extended result in rd

B.103.3. Access

M	S	U
Always	Always	Always

B.103.4. Decode Variables

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

B.103.5. Execution

```
Bits<32> t0 = X[rs1][31:0];
Bits<32> t1 = X[rs2][31:0];
X[rd] = sext(t0 - t1, 31);
```

B.103.6. Exceptions

This instruction does not generate synchronous exceptions.

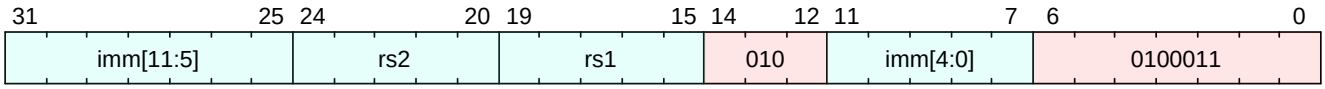
B.104. sw

Store word

This instruction is defined by:

- I, version >= 0

B.104.1. Encoding



B.104.2. Synopsis

Store 32 bits of data from register `rs2` to an address formed by adding `rs1` to a signed offset.

B.104.3. Access

M	S	U
Always	Always	Always

B.104.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

B.104.5. Execution

```
XReg virtual_address = X[rs1] + imm;
write_memory<32>(virtual_address, X[rs2][31:0], $encoding);
```

B.104.6. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

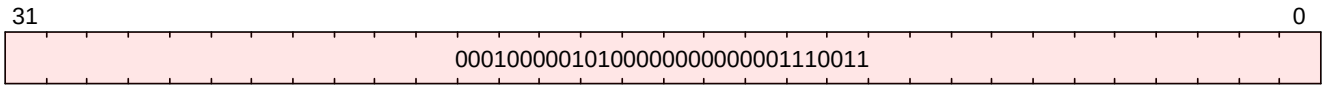
B.105. wfi

Wait for interrupt

This instruction is defined by:

- Sm, version >= 0

B.105.1. Encoding



B.105.2. Synopsis

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

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

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

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

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

<%- end -%>

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

B.105.3. 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.105.4. Decode Variables

```
```

B.105.5. Execution

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

```
      raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
    }
  }
}
wfi();
```

B.105.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

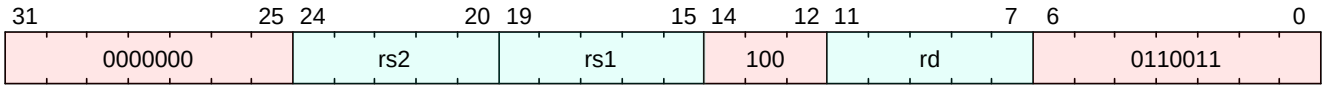
B.106. xor

Exclusive Or

This instruction is defined by:

- I, version >= 0

B.106.1. Encoding



B.106.2. Synopsis

Exclusive or rs1 with rs2, and store the result in rd

B.106.3. Access

M	S	U
Always	Always	Always

B.106.4. Decode Variables

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

B.106.5. Execution

```
X[rd] = X[rs1] ^ X[rs2];
```

B.106.6. Exceptions

This instruction does not generate synchronous exceptions.

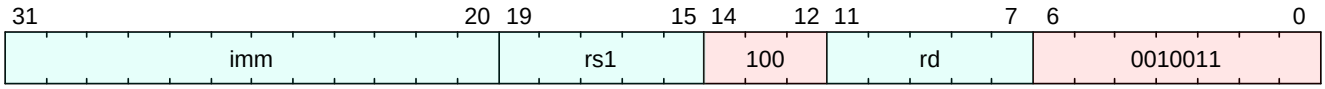
B.107. xori

Exclusive Or immediate

This instruction is defined by:

- I, version >= 0

B.107.1. Encoding



B.107.2. Synopsis

Exclusive or an immediate to the value in rs1, and store the result in rd

B.107.3. Access

M	S	U
Always	Always	Always

B.107.4. Decode Variables

```
Bits<12> imm = $encoding[31:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

B.107.5. Execution

```
X[rd] = X[rs1] ^ imm;
```

B.107.6. 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 >= 0
Length	64-bit
Privilege Mode	U

C.1.2. Format

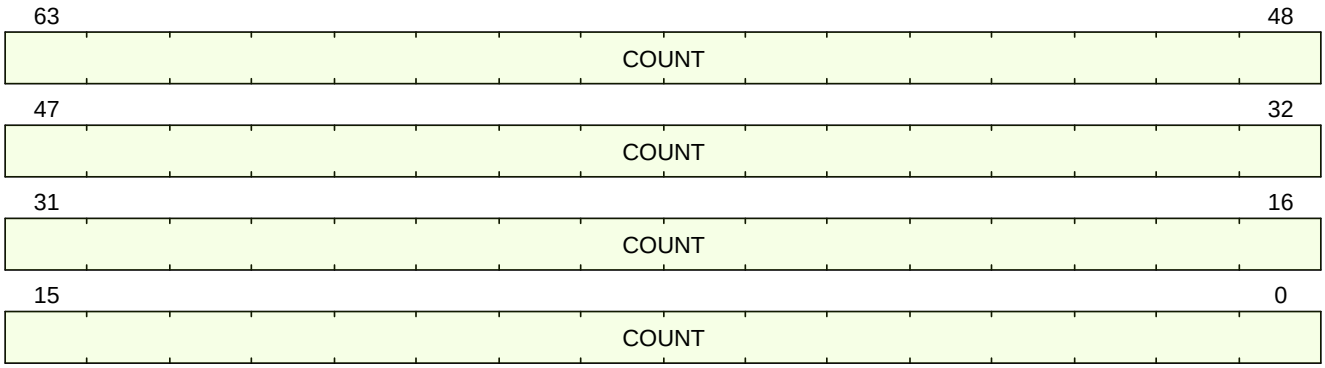


Figure 1. cycle format

C.1.3. Field Summary

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

C.1.4. Fields

COUNT

Location

63:0

Description

Alias of mcycle.COUNT.

Type

RO-H

Reset value

UNDEFINED_LEGAL

C.1.5. Software read

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

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

```
  } else if (CSR[mcounteren].CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle();
```

C.2. cycleh

High-half cycle counter for RDCYCLE Instruction

> **NOTE** 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 >= 0
Length	32-bit
Privilege Mode	U

C.2.2. Format

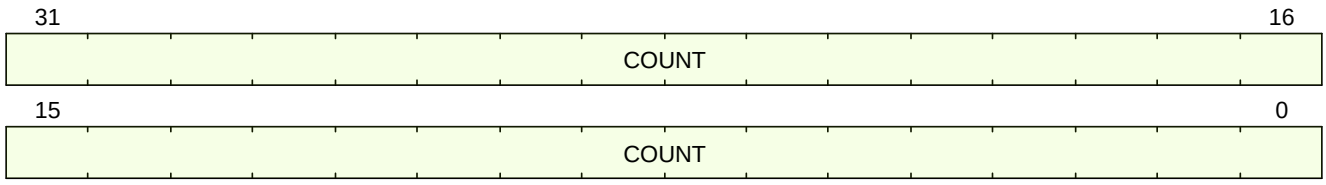


Figure 2. cycleh format

C.2.3. Field Summary

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

C.2.4. Fields

COUNT

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

```
  } else if (CSR[mcounteren].CY == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_mcycle()[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 >= 0
Length	64-bit
Privilege Mode	U

C.3.2. Format

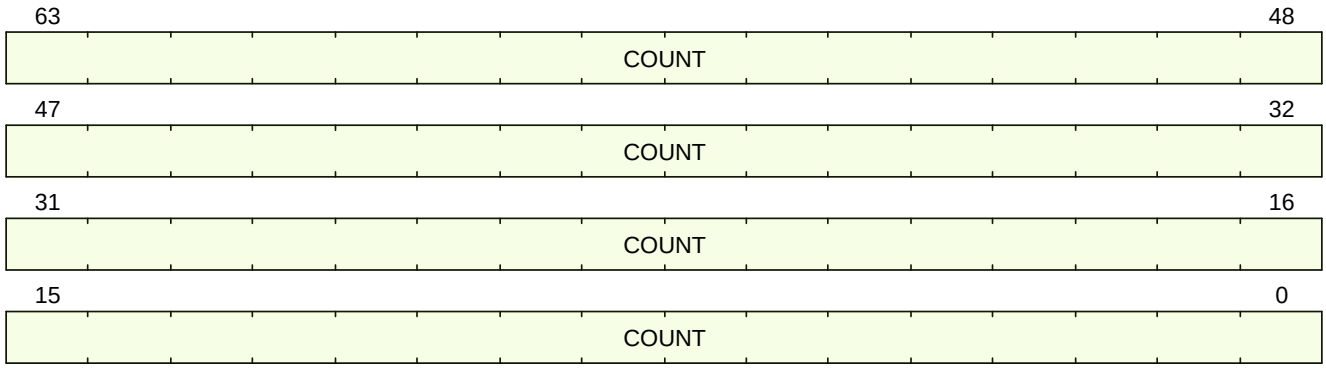


Figure 3. instret format

C.3.3. Field Summary

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

C.3.4. Fields

COUNT

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

```
  } else if (CSR[mcounteren].IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return CSR[minstret].COUNT;
```

C.4. instreth

Instructions retired counter, high bits

> **NOTE** 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 >= 0
Length	32-bit
Privilege Mode	U

C.4.2. Format

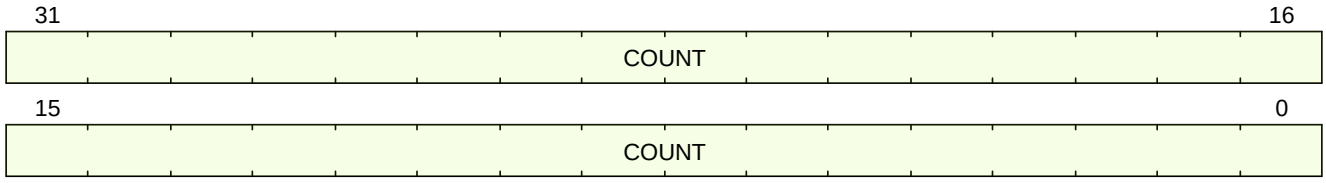


Figure 4. instreth format

C.4.3. Field Summary

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

C.4.4. Fields

COUNT

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

```
  } else if (CSR[mcounteren].IR == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return 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.

NOTE

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 >= 0
Length	64-bit
Privilege Mode	M

C.5.2. Format

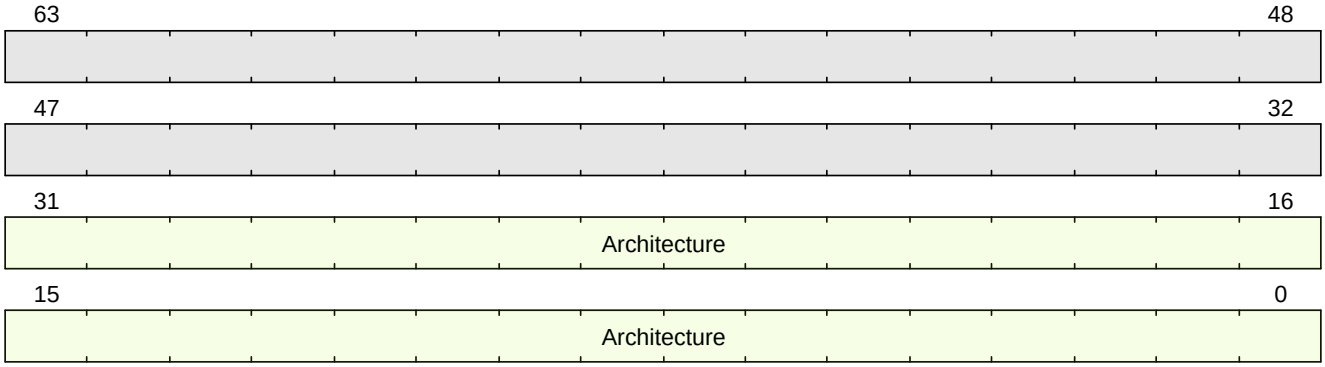


Figure 5. marchid format

C.5.3. Field Summary

Name	Location	Type	Reset Value
Architecture	63:0	RO	ARCH_ID

C.5.4. Fields

Architecture

Location

63:0

Description

Vendor-specific microarchitecture ID.

Type

RO

Reset value

ARCH_ID

C.6. mcause

Machine Cause

Reports the cause of the latest exception.

C.6.1. Attributes

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

C.6.2. Format

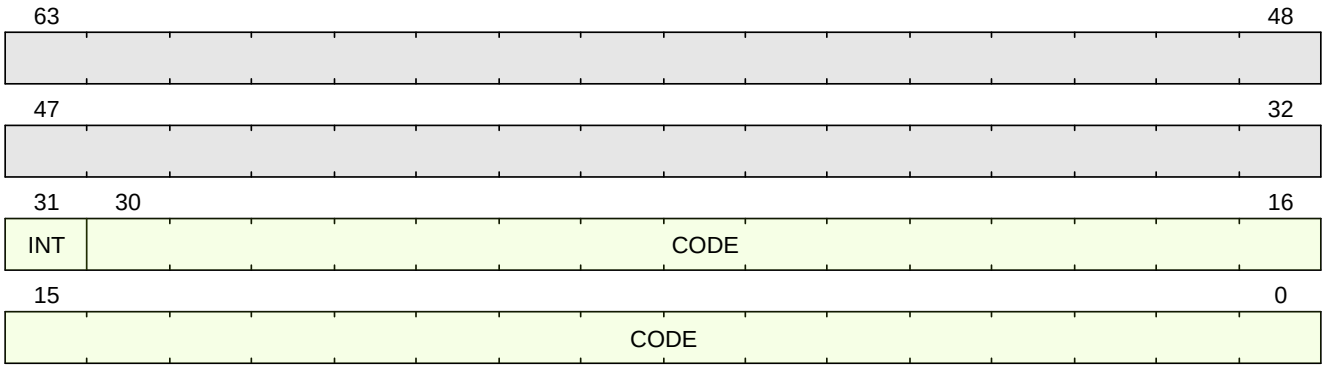


Figure 6. mcause format

C.6.3. Field Summary

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

C.6.4. Fields

INT

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 writeable.

[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

CODE

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 writeable.

[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. mconfigptr

Machine Configuration Pointer

Holds a physical address pointer to the unified discovery data structure in Memory.

The mconfigptr holds the physical address of a configuration data structure. Software can traverse this data structure to discover information about the harts, the platform, and their configuration.

The pointer alignment in bits must be no smaller than MXLEN: i.e., if MXLEN is $8\times n$, then mconfigptr[$\log_2 n$-1:0] must be zero.

The mconfigptr register must be implemented, but it may be zero to indicate the configuration data structure does not exist or that an alternative mechanism must be used to locate it.

The format and schema of the configuration data structure have yet to be standardized.

NOTE

While the mconfigptr register will simply be hardwired in some implementations, other implementations may provide a means to configure the value returned on CSR reads. For example, mconfigptr might present the value of a memory-mapped register that is programmed by the platform or by M-mode software towards the beginning of the boot process.

C.7.1. Attributes

CSR Address	0xf15
Defining extension	• Sm, version >=1.12
Length	64-bit
Privilege Mode	M

C.7.2. Format

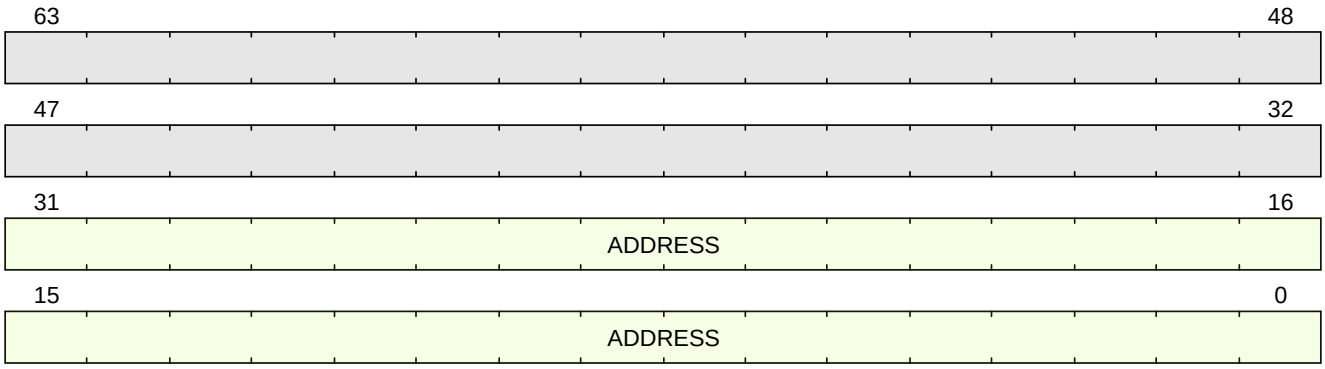


Figure 7. mconfigptr format

C.7.3. Field Summary

Name	Location	Type	Reset Value
ADDRESS	63:0	RO	CONFIG_PTR_ADDRESS

C.7.4. Fields

ADDRESS

Location

63:0

Description

Pointer to physical address of the Unified Discovery configuration data structure.

Type

RO

Reset value

CONFIG_PTR_ADDRESS

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.

NOTE

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 >= 0 ◦ Smhpm, version >= 0
Length	32-bit
Privilege Mode	M

C.8.2. Format

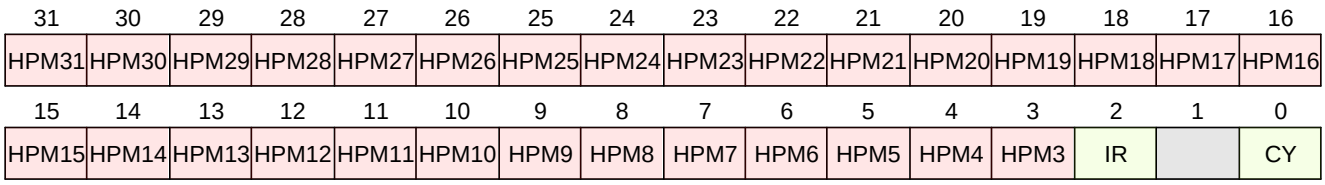


Figure 8. mcountinhibit format

C.8.3. Field Summary

Name	Location	Type	Reset Value
CY	0	RW RO	UNDEFINED_LEGAL 0
IR	2	RW RO	UNDEFINED_LEGAL 0
HPM3	3	RW RO	UNDEFINED_LEGAL 0
HPM4	4	RW RO	UNDEFINED_LEGAL 0
HPM5	5	RW RO	UNDEFINED_LEGAL 0
HPM6	6	RW RO	UNDEFINED_LEGAL 0
HPM7	7	RW RO	UNDEFINED_LEGAL 0
HPM8	8	RW RO	UNDEFINED_LEGAL 0
HPM9	9	RW RO	UNDEFINED_LEGAL 0
HPM10	10	RW RO	UNDEFINED_LEGAL 0
HPM11	11	RW RO	UNDEFINED_LEGAL 0
HPM12	12	RW RO	UNDEFINED_LEGAL 0
HPM13	13	RW RO	UNDEFINED_LEGAL 0

Name	Location	Type	Reset Value
HPM14	14	RW RO	UNDEFINED_LEGAL 0
HPM15	15	RW RO	UNDEFINED_LEGAL 0
HPM16	16	RW RO	UNDEFINED_LEGAL 0
HPM17	17	RW RO	UNDEFINED_LEGAL 0
HPM18	18	RW RO	UNDEFINED_LEGAL 0
HPM19	19	RW RO	UNDEFINED_LEGAL 0
HPM20	20	RW RO	UNDEFINED_LEGAL 0
HPM21	21	RW RO	UNDEFINED_LEGAL 0
HPM22	22	RW RO	UNDEFINED_LEGAL 0
HPM23	23	RW RO	UNDEFINED_LEGAL 0
HPM24	24	RW RO	UNDEFINED_LEGAL 0
HPM25	25	RW RO	UNDEFINED_LEGAL 0
HPM26	26	RW RO	UNDEFINED_LEGAL 0
HPM27	27	RW RO	UNDEFINED_LEGAL 0

Name	Location	Type	Reset Value
HPM28	28	RW RO	UNDEFINED_LEGAL 0
HPM29	29	RW RO	UNDEFINED_LEGAL 0
HPM30	30	RW RO	UNDEFINED_LEGAL 0
HPM31	31	RW RO	UNDEFINED_LEGAL 0

C.8.4. Fields

CY

Location

0

Description

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

Type

RW

RO

Reset value

UNDEFINED_LEGAL

0

IR

Location

2

Description

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

Type

RW

RO

Reset value

UNDEFINED_LEGAL

0

HPM3

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

0

HPM4

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

0

HPM5

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

0

HPM6

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

0

HPM7

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

0

HPM8

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

0

HPM9

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

0

HPM10

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

0

HPM11

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

0

HPM12

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

0

HPM13

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

0

HPM14

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

0

HPM15

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

0

HPM16

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

0

HPM17

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

0

HPM18

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

0

HPM19

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

0

HPM20

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

0

HPM21

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

0

HPM22

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

0

HPM23

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

0

HPM24

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

0

HPM25

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

0

HPM26

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

0

HPM27

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

0

HPM28

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

0

HPM29

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

0

HPM30

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

0

HPM31

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

0

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 >= 0
Length	64-bit
Privilege Mode	M

C.9.2. Format

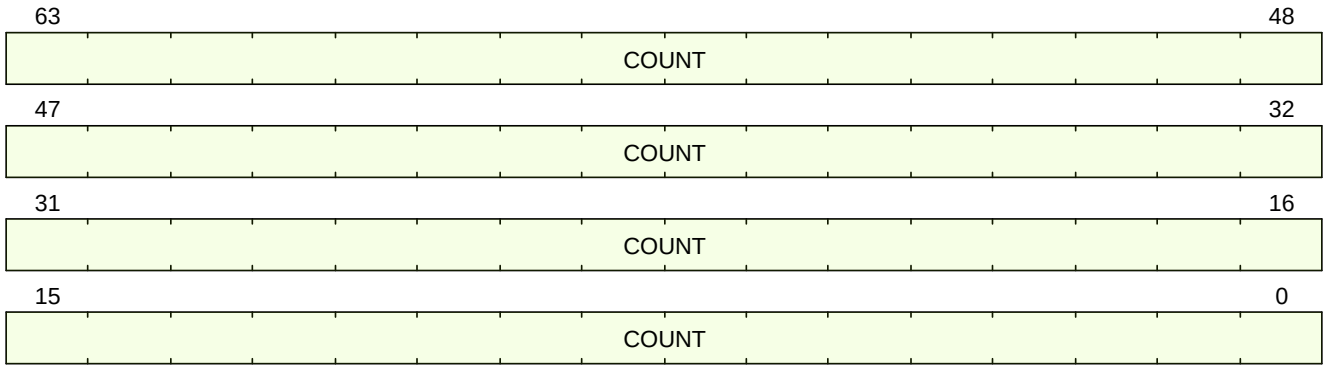


Figure 9. mcycle format

C.9.3. Field Summary

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

C.9.4. Fields

COUNT

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

> **NOTE** mcycleh is only defined in RV32.

High-half alias of mcycle.

C.10.1. Attributes

CSR Address	0xb80
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	M

C.10.2. Format

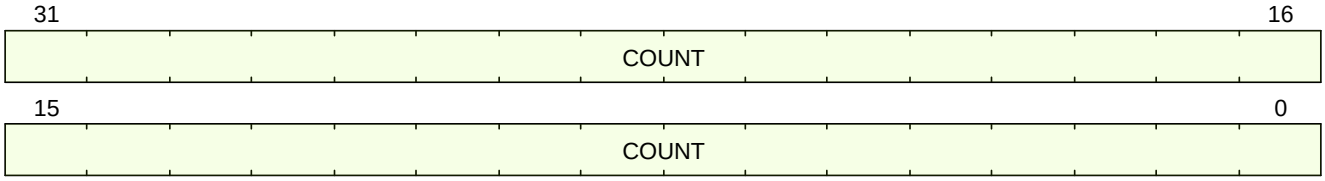


Figure 10. mcycleh format

C.10.3. Field Summary

Name	Location	Type	Reset Value
COUNT	31:0	RW-RH	UNDEFINED_LEGAL

C.10.4. Fields

COUNT

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. 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.11.1. Attributes

CSR Address	0x341
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.11.2. Format

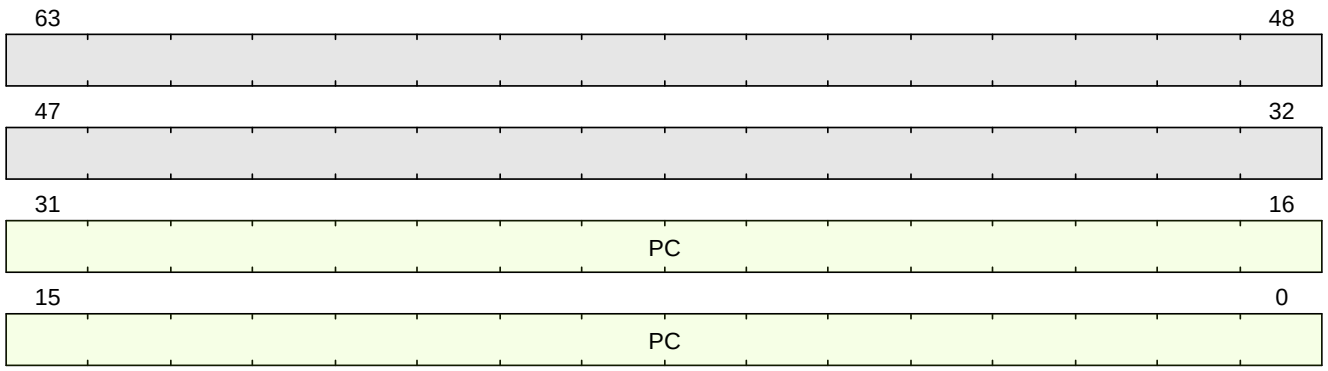


Figure 11. mepc format

C.11.3. Field Summary

Name	Location	Type	Reset Value
PC	63:0	RW-RH	0

C.11.4. Fields

PC

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.11.5. 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.11.6. Software read

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

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

C.12. mhartid

Machine Hart ID

Reports the unique hart-specific ID in the system.

C.12.1. Attributes

CSR Address	0xf14
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.12.2. Format

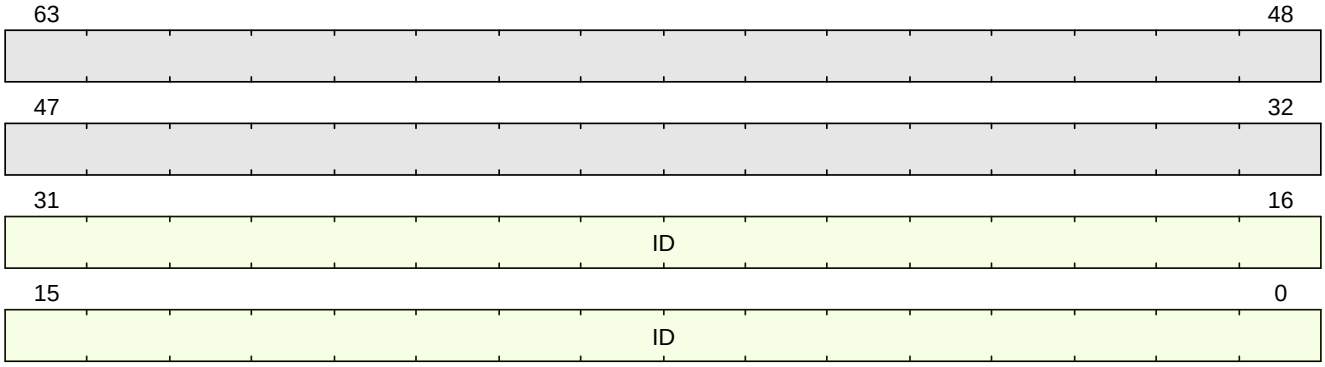


Figure 12. mhartid format

C.12.3. Field Summary

Name	Location	Type	Reset Value
ID	63:0	RO	UNDEFINED_LEGAL

C.12.4. Fields

ID

Location

63:0

Description

hart-specific ID.

Type

RO

Reset value

UNDEFINED_LEGAL

C.12.5. Software read

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

```
return hartid();
```

C.13. mie

Machine Interrupt Enable

{"$copy"⇒"mip.yaml#/description"}

C.13.1. Attributes

CSR Address	0x304
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.13.2. Format

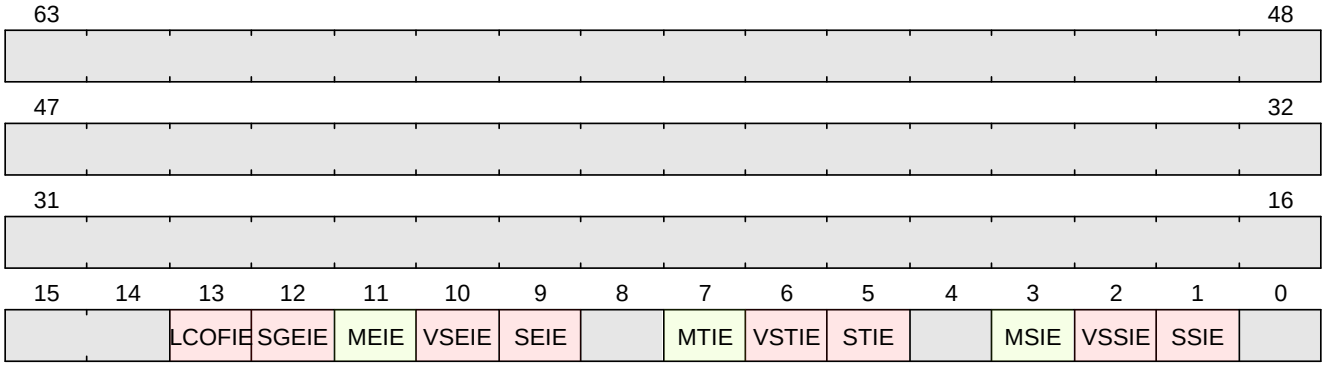


Figure 13. mie format

C.13.3. Field Summary

Name	Location	Type	Reset Value
SSIE	1	RW	0
VSSIE	2	RW	0
MSIE	3	RW	0
STIE	5	RW	0
VSTIE	6	RW	0
MTIE	7	RW	0
SEIE	9	RW	0

Name	Location	Type	Reset Value
VSEIE	10	RW	0
MEIE	11	RW	0
SGEIE	12	RW	0
LCOFIE	13	RW	0

C.13.4. Fields

SSIE

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

VSSIE

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

MSIE

Location

3

Description

Enables Machine Software Interrupts.

Type

RW

Reset value

0

STIE

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

VSTIE

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

MTIE

Location

7

Description

Enables Machine Timer Interrupts.

Type

RW

Reset value

0

SEIE

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

VSEIE

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

MEIE

Location

11

Description

Enables Machine External Interrupts.

Type

RW

Reset value

0

SGEIE

Location

12

Description

Enables Supervisor Guest External Interrupts

Alias of `hie.SGEIE`.

Type

RW

Reset value

0

LCOFIE

Location

13

Description

Enables Local Counter Overflow Interrupts.

Alias of `sie.LCOFIE` when mideleg.LCOFI is set. Otherwise, `sie.LCOFIE` is an independent writeable 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.14. 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.

NOTE 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.14.1. Attributes

CSR Address	0xf13
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.14.2. Format

Figure 14. mimpid format

C.14.3. Field Summary

Name	Location	Type	Reset Value
Implementation	63:0	RO	IMP_ID

C.14.4. Fields

`Implementation`

Location

63:0

Description

Vendor-specific implementation ID.

Type

RO

Reset value

IMP_ID

C.15. minstret

Machine Instructions Retired Counter

Counts the number of instructions retired by this hart from some arbitrary start point in the past.

NOTE	Instructions that cause synchronous exceptions, including ecall and ebreak, are not considered to retire and hence do not increment the minstret CSR.

C.15.1. Attributes

CSR Address	0xb02
Defining extension	• Zicntr, version >= 0
Length	64-bit
Privilege Mode	M

C.15.2. Format

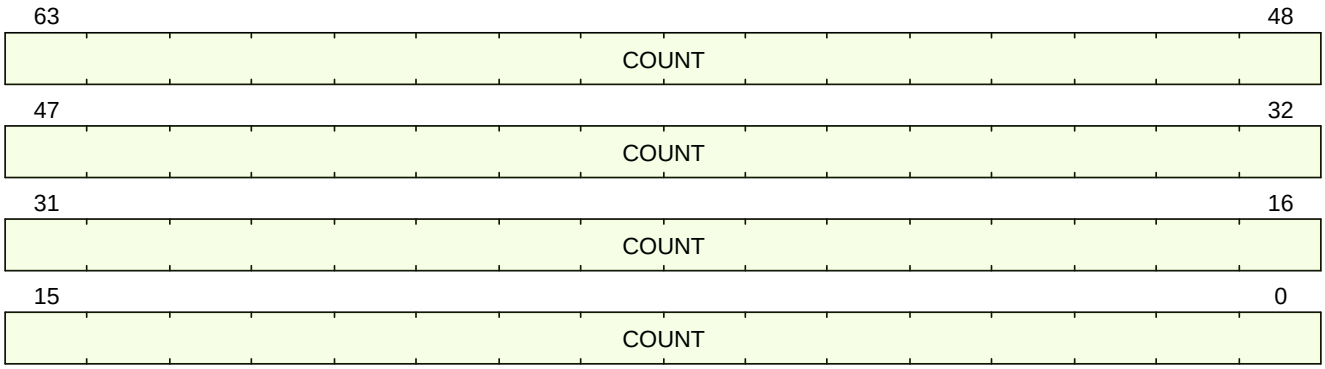


Figure 15. minstret format

C.15.3. Field Summary

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

C.15.4. Fields

COUNT

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.16. minstreth

Machine Instructions Retired Counter

Upper half of 64-bit instructions retired counters.

See minstret for details.

C.16.1. Attributes

CSR Address	0xb02
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	M

C.16.2. Format

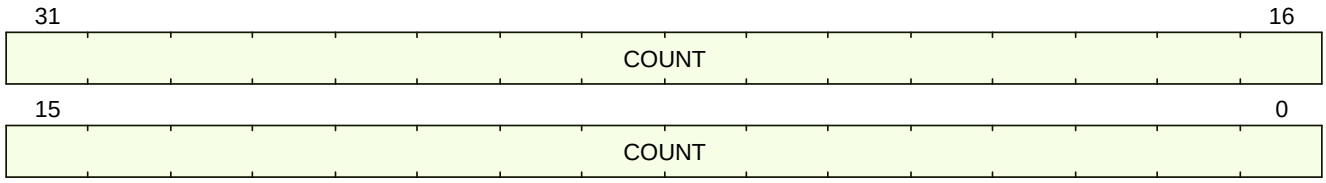


Figure 16. minstreth format

C.16.3. Field Summary

Name	Location	Type	Reset Value
COUNT	31:0	RW-H	UNDEFINED_LEGAL

C.16.4. Fields

COUNT

Location

31:0

Description

Instructions retired counter

Upper half of minstret.

Type

RW-H

Reset value

UNDEFINED_LEGAL

C.16.5. 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.16.6. Software read

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

```
return CSR[minstret].COUNT[63:32];
```

C.17. 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.

NOTE 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.

NOTE 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.

NOTE

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.

NOTE

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.17.1. Attributes

CSR Address	0x344
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.17.2. Format

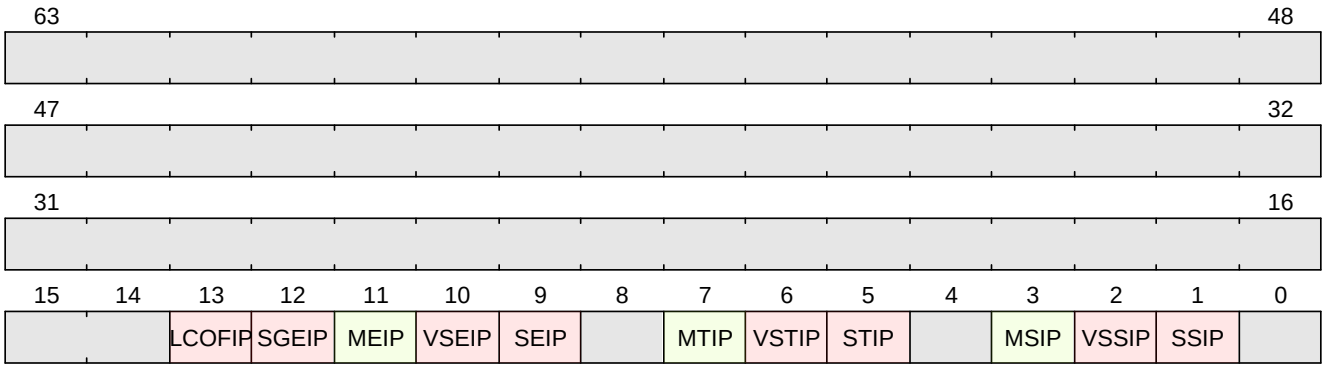


Figure 17. mip format

C.17.3. Field Summary

Name	Location	Type	Reset Value
SSIP	1	RW	0
VSSIP	2	RW	0

Name	Location	Type	Reset Value
MSIP	3	RO	0
STIP	5	RW	0
VSTIP	6	RO-H	0
MTIP	7	RO-H	0
SEIP	9	RW-H	0
VSEIP	10	RO-H	0
MEIP	11	RO-H	0
SGEIP	12	RO-H	0
LCOFIP	13	RW-H	0

C.17.4. Fields

SSIP

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

VSSIP

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

MSIP

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

STIP

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

VSTIP

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 writeable)

Type

RO-H

Reset value

0

MTIP

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 writeable.

Type

RO-H

Reset value

0

SEIP

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-writeable 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-writeable 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-writeable 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

VSEIP

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

MEIP

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

SGEIP

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

LCOFIP

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.18. misa

Machine ISA Control

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

C.18.1. Attributes

CSR Address	0x301
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.18.2. Format

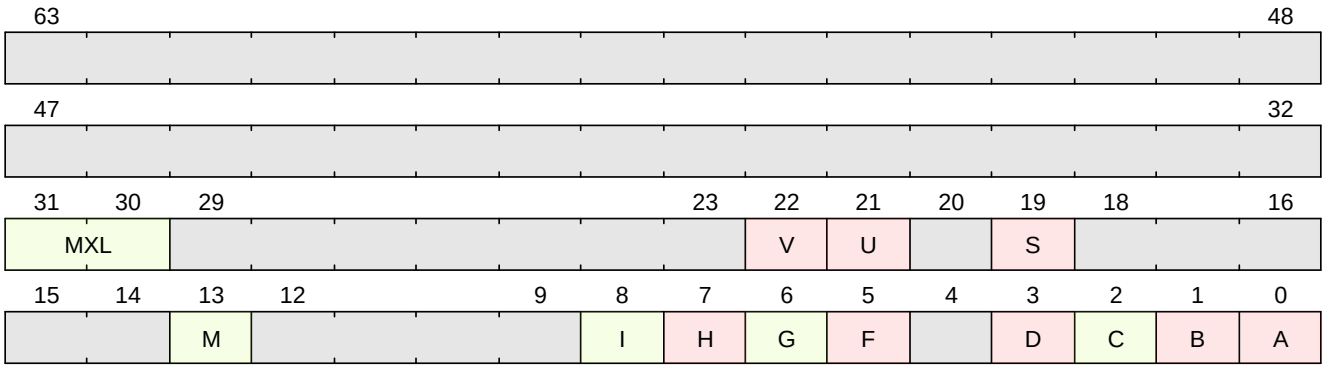


Figure 18. misa format

C.18.3. Field Summary

Name	Location	Type	Reset Value
MXL	63:62	RO	2
A	0	RW RO	1
B	1	RW RO	1
C	2	RW RO	1
D	3	RW RO	1

Name	Location	Type	Reset Value
F	5	RW RO	1
G	6	RO-H RO	1 0
H	7	RW RO	1
I	8	RO	1
M	13	RW RO	1
S	19	RW RO	1
U	21	RW RO	1
V	22	RO	1

C.18.4. Fields

MXL

Location
63:62

Description
XLEN in M-mode.

Type
RO

Reset value
2

A

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

1

B

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

C

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

D

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

1

F

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

1

G

Location

6

Description

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

Type

RO-H

RO

Reset value

1

0

H

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

1

I

Location

8

Description

Indicates support for the I (base) extension.

Type

RO

Reset value

1

M

Location

13

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

S

Location

19

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

U

Location

21

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

V

Location

22

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

RO

Reset value

1

C.18.5. 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
S = csr_value.S
U = csr_value.U
V = csr_value.V
```

C.18.6. Software read

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

```
return ((CSR[misa].MXL << 62) | (CSR[misa].V << 21) | (CSR[misa].U << 20) |
(CSR[misa].S << 18) | (CSR[misa].M << 12) | (CSR[misa].I << 7) | (CSR[misa].H << 6) |
((CSR[misa].A & CSR[misa].M & CSR[misa].F & CSR[misa].D) << 5) | (CSR[misa].F << 4) |
(CSR[misa].D << 3) | (CSR[misa].C << 2) | (CSR[misa].B << 1) | CSR[misa].A);
```

C.19. 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.19.1. Attributes

CSR Address	0x741
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.19.2. Format

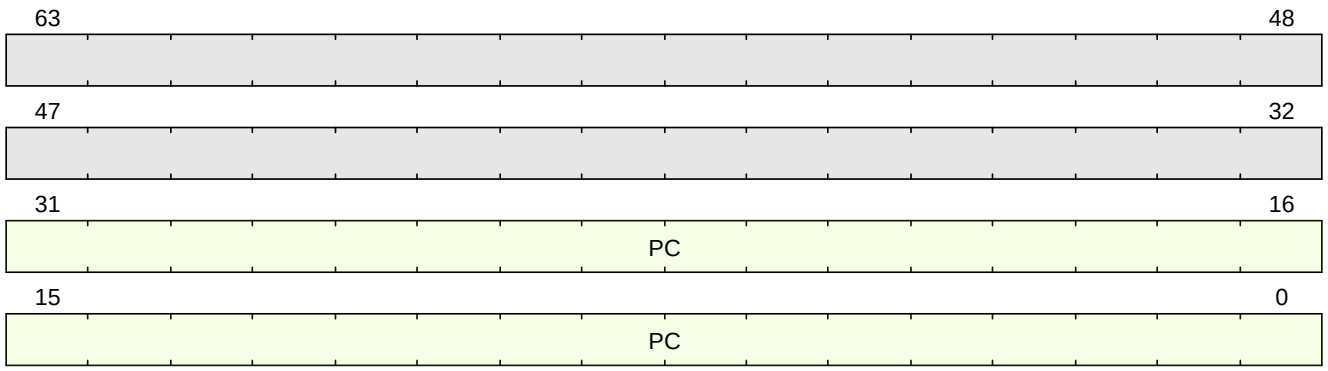


Figure 19. mnepc format

C.19.3. Field Summary

Name	Location	Type	Reset Value
PC	63:0	RW-RH	UNDEFINED_LEGAL

C.19.4. Fields

PC

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.19.5. 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.19.6. Software read

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

```
if (implemented?(ExtensionName::C) && CSR[misa].C == 1'b1) {
  return CSR[mnepc].PC & ~64'b1;
} else {
  return CSR[mnepc].PC & ~64'b11;
}
```

C.20. mscratch

Machine Scratch Register

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

C.20.1. Attributes

CSR Address	0x340
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.20.2. Format

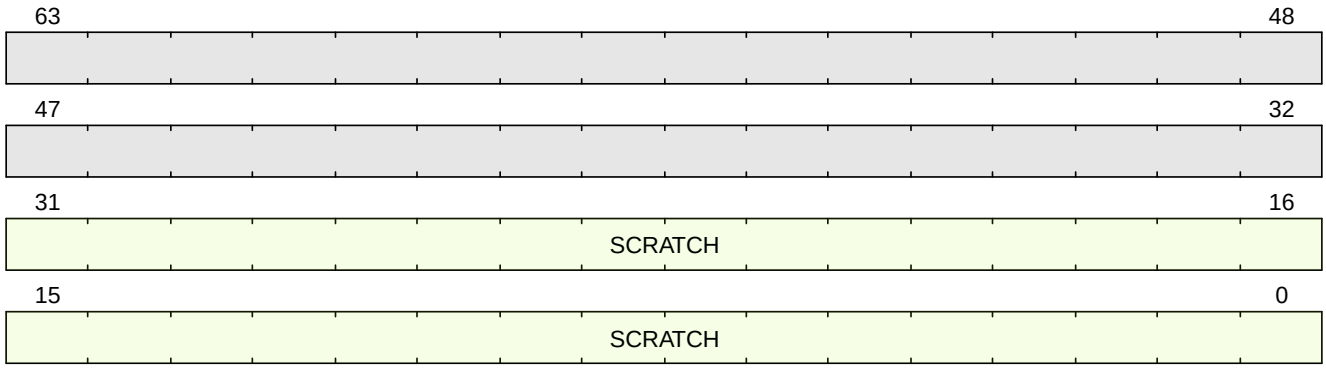


Figure 20. mscratch format

C.20.3. Field Summary

Name	Location	Type	Reset Value
SCRATCH	63:0	RW	0

C.20.4. Fields

SCRATCH

Location

63:0

Description

Scratch value

Type

RW

Reset value

0

C.21. mseccfg

Machine Security Configuration

Machine Security Configuration

C.21.1. Attributes

CSR Address	0x747
Defining extension	• Sm, version >= 1.12
Length	64-bit
Privilege Mode	M

C.21.2. Format

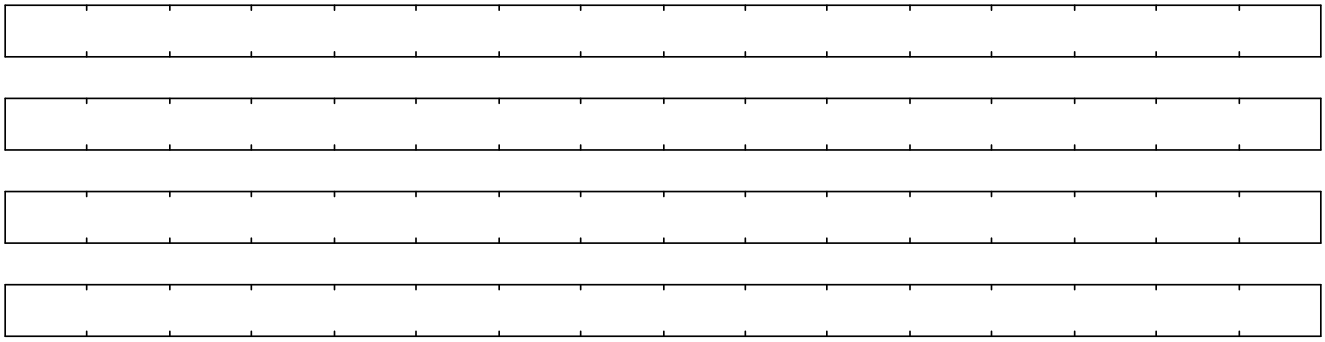


Figure 21. mseccfg format

C.21.3. Field Summary

Name	Location	Type	Reset Value

C.21.4. Fields

This CSR has no fields. However, it must still exist (not cause an `Illegal Instruction` trap) and always return zero on a read.

C.22. mseccfgh

Most significant 32 bits of Machine Security Configuration

Machine Security Configuration

C.22.1. Attributes

CSR Address	0x757
Defining extension	• Sm, version >= 1.12
Length	32-bit
Privilege Mode	M

C.22.2. Format

Figure 22. mseccfgh format

C.22.3. Field Summary

Name	Location	Type	Reset Value

C.22.4. Fields

This CSR has no fields. However, it must still exist (not cause an `Illegal Instruction` trap) and always return zero on a read.

C.23. mstatus

Machine Status

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

C.23.1. Attributes

CSR Address	0x300
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.23.2. Format

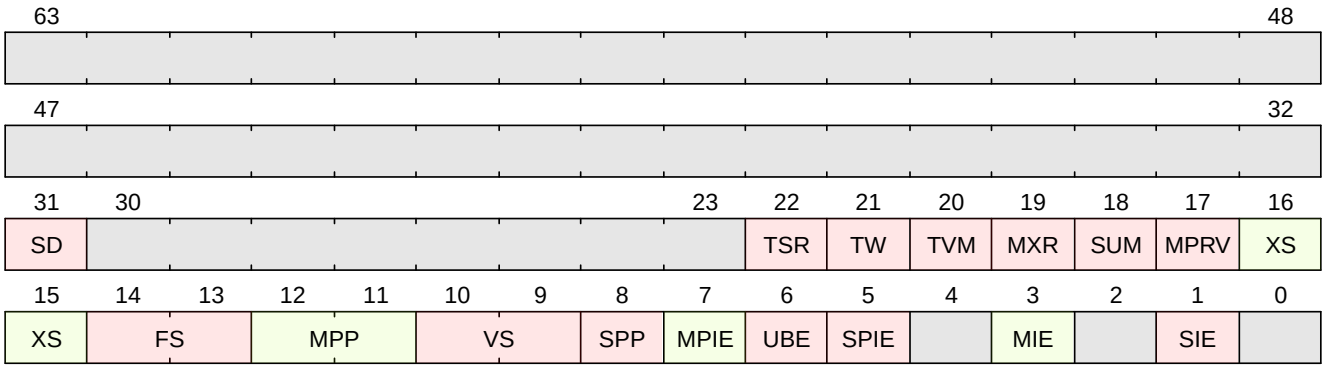


Figure 23. mstatus format

C.23.3. Field Summary

Name	Location	Type	Reset Value
SD	63	RO-H RO-H RO	mstatus_sd_reset_value() UNDEFINED_LEGAL
MDT	42	RW-H	UNDEFINED_LEGAL
MPV	39	RW-H	UNDEFINED_LEGAL
GVA	38	RW-H	0
MBE	37	RO	0
SBE	36	RW RO	0 1 UNDEFINED_LEGAL

Name	Location	Type	Reset Value
SXL	35:34	RW RO	UNDEFINED_LEGAL
UXL	33:32	RW RO	0 1 UNDEFINED_LEGAL
TSR	22	RW	UNDEFINED_LEGAL
TW	21	RW	UNDEFINED_LEGAL
TVM	20	RO RW	0 UNDEFINED_LEGAL 0
MXR	19	RW	UNDEFINED_LEGAL
SUM	18	RW RO	UNDEFINED_LEGAL 0
MPRV	17	RW-H RO	0
XS	16:15	RO	0
FS	14:13	RW-H RO RW RO	UNDEFINED_LEGAL 0 UNDEFINED_LEGAL 0
MPP	12:11	RW-H	UNDEFINED_LEGAL
VS	10:9	RW-H	UNDEFINED_LEGAL
SPP	8	RW-H	UNDEFINED_LEGAL
MPIE	7	RW-H	UNDEFINED_LEGAL
UBE	6	RW RO	0 1 UNDEFINED_LEGAL
SPIE	5	RW-H RO	UNDEFINED_LEGAL 0

Name	Location	Type	Reset Value
MIE	3	RW-H	0
SIE	1	RW-H RO	UNDEFINED_LEGAL 0

C.23.4. Fields

SD

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

mstatus_sd_reset_value()

UNDEFINED_LEGAL

MDT

IMPORTANT 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

MPV

IMPORTANT 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

GVA

IMPORTANT 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

MBE

IMPORTANT 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_ENDIANESS == little"]
Since the CPU does not support big endian, this is hardwired to 0.

[when,"M_MODE_ENDIANESS == big"]
Since the CPU does not support little endian, this is hardwired to 1.

Type

RO

Reset value

0

SBE

IMPORTANT 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_ENDIANESS == little"]

Since the CPU does not support big endian, this is hardwired to 0.

[when,"S_MODE_ENDIANESS == big"]
Since the CPU does not support little endian, this is hardwired to 1.

Type

RW

RO

Reset value

0

1

UNDEFINED_LEGAL

SXL

IMPORTANT	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

UXL

IMPORTANT	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

0

1

UNDEFINED_LEGAL

TSR

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

TW

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

TVM

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

0

UNDEFINED_LEGAL

0

MXR

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

SUM

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

0

MPRV

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

XS

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

FS

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

0

UNDEFINED_LEGAL

0

MPP

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

VS

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

Reset value

UNDEFINED_LEGAL

SPP

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

MPIE

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

UBE

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

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

Type

RW

RO

Reset value

0

1

UNDEFINED_LEGAL

SPIE

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

0

MIE

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

SIE

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

```
0
```

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:

```
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.24. mstatush

Machine Status High

NOTE	mstatush is only defined in RV32.

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

C.24.1. Attributes

CSR Address	0x310
Defining extension	• Sm, version >= 1.12
Length	32-bit
Privilege Mode	M

C.24.2. Format

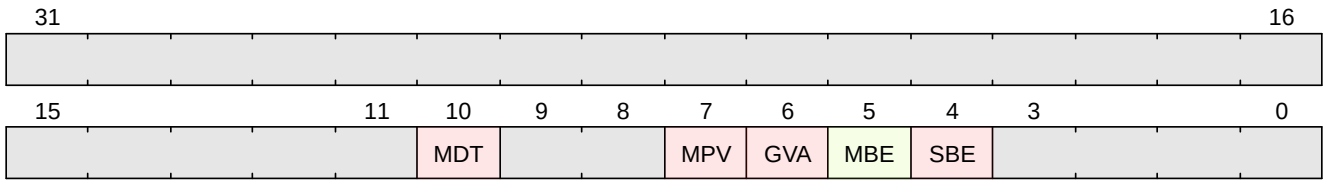


Figure 24. mstatush format

C.24.3. Field Summary

Name	Location	Type	Reset Value
MDT	10	RW-H	UNDEFINED_LEGAL
MPV	7	RW-H	UNDEFINED_LEGAL
GVA	6	RW-H	0
MBE	5	RO	0
SBE	4	RW RO	0 1 UNDEFINED_LEGAL

C.24.4. Fields

MDT

Location

10

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 mstatush.MDT=1, direct write by CSR instruction cannot set mstatus.MIE to 1.

Type

RW-H

Reset value

UNDEFINED_LEGAL

MPV

Location

7

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

GVA

Location

6

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

MBE

Location

5

Description

see mstatus.MBE

Type

RO

Reset value

0

SBE

Location

4

Description

see mstatus.SBE

Type

RW

RO

Reset value

0

1

UNDEFINED_LEGAL

C.25. mtval

Machine Trap Value

Holds trap-specific information

C.25.1. Attributes

CSR Address	0x343
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.25.2. Format

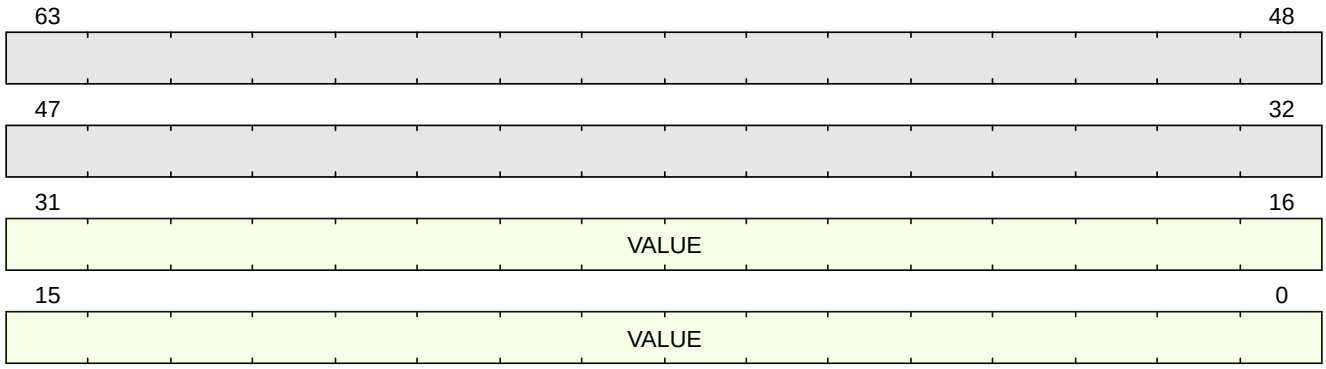


Figure 25. mtval format

C.25.3. Field Summary

Name	Location	Type	Reset Value
VALUE	63:0	RW-H	0

C.25.4. Fields

VALUE

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.26. mtvec

Machine Trap Vector Control

Controls where traps jump.

C.26.1. Attributes

CSR Address	0x305
Defining extension	• Sm, version >= 0
Length	64-bit
Privilege Mode	M

C.26.2. Format

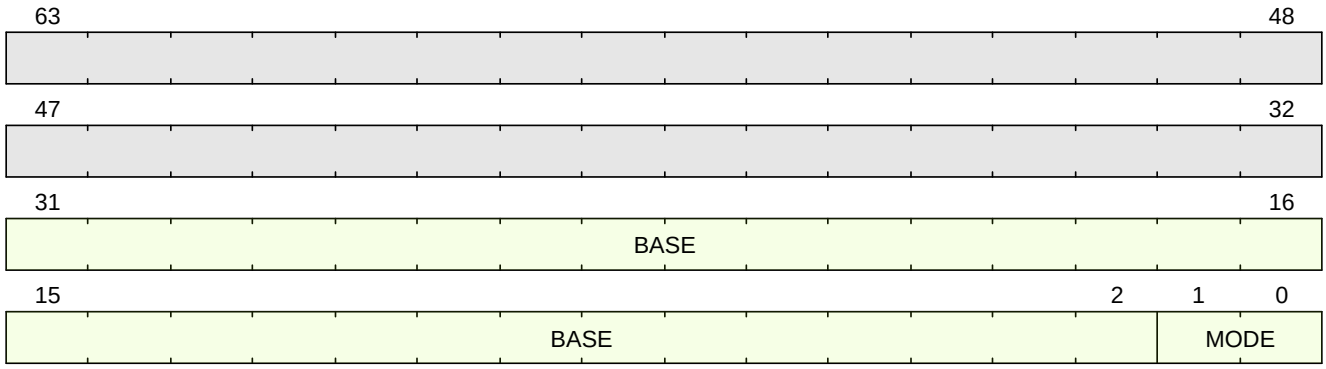


Figure 26. mtvec format

C.26.3. Field Summary

Name	Location	Type	Reset Value
BASE	63:2	RW-R	0
MODE	1:0	RW-R	UNDEFINED_LEGAL

C.26.4. Fields

BASE

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

MODE

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.26.5. 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.27. mvendorid

Machine Vendor ID

Reports the JEDEC manufacturer ID of the core.

C.27.1. Attributes

CSR Address	0xf11
Defining extension	• Sm, version >= 0
Length	32-bit
Privilege Mode	M

C.27.2. Format

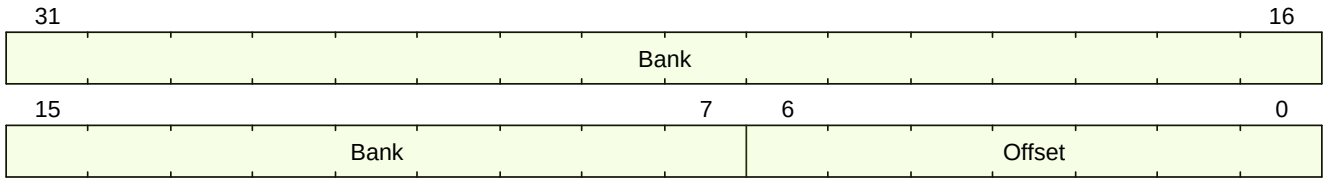


Figure 27. mvendorid format

C.27.3. Field Summary

Name	Location	Type	Reset Value
Bank	31:7	RO	VENDOR_ID_BANK
Offset	6:0	RO	VENDOR_ID_OFFSET

C.27.4. Fields

Bank

Location

31:7

Description

JEDEC manufacturer ID bank minus 1

Type

RO

Reset value

VENDOR_ID_BANK

`Offset`

Location

6:0

Description

JEDEC manufacturer ID offset

Type

RO

Reset value

VENDOR_ID_OFFSET

C.28. 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.28.1. Attributes

CSR Address	0xc01
Defining extension	• Zicntr, version >= 0
Length	64-bit
Privilege Mode	U

C.28.2. Format

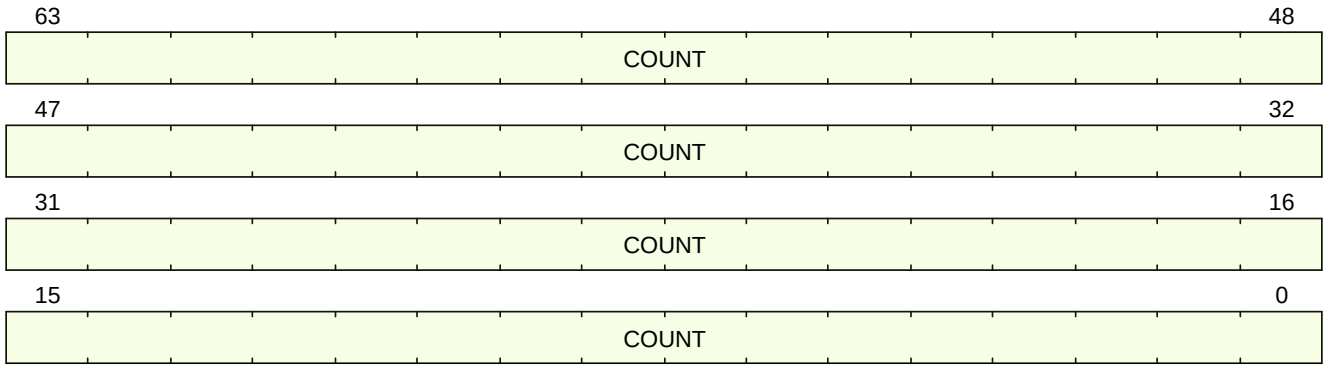


Figure 28. time format

C.28.3. Field Summary

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

C.28.4. Fields

`COUNT`

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.28.5. Software read

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

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

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

C.29. timeh

High-half timer for RDTIME Instruction

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.29.1. Attributes

CSR Address	0xc81
Defining extension	• Zicntr, version >= 0
Length	32-bit
Privilege Mode	U

C.29.2. Format

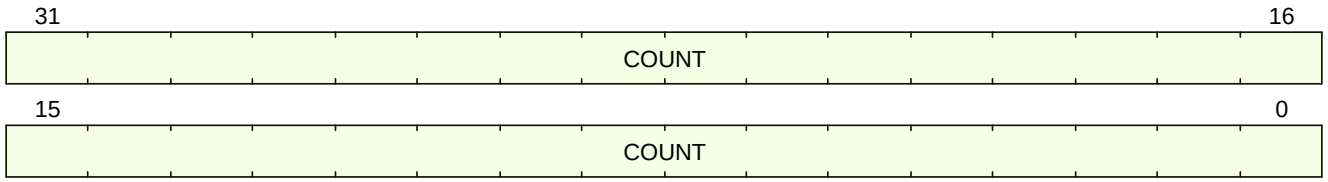


Figure 29. timeh format

C.29.3. Field Summary

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

C.29.4. Fields

COUNT

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.29.5. Software read

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

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

```
}
return read_mtime()[63:32];
```