# RVI20 Profile Release

## Table of Contents



TODO: revmark

## Copyright and license information



## Acknowledgements

Contributors to the RVI20 Profile (in alphabetical order) include:

- Krste Asanovic <krste@sifive.com> (SiFive)

We express our gratitude to everyone that contributed to, reviewed or improved this specification through their comments and questions.

## 1. RISC-V Profiles

RISC-V was designed to provide a highly modular and extensible instruction set, and includes a large and growing set of standard extensions. In addition, users may add their own custom extensions. This flexibility can be used to highly optimize a specialized design by including only the exact set of ISA features required for an application, but the same flexibility also leads to a combinatorial explosion in possible ISA choices. Profiles specify a much smaller common set of ISA choices that capture the most value for most users, and which thereby enable the software community to focus resources on building a rich software ecosystem with application and operating system portability across different implementations.

*Another pragmatic concern is the long and unwieldy ISA strings required to encode common sets of extensions, which will continue to grow as new extensions are defined.*

Each profile is built on a standard base ISA plus a set of mandatory ISA extensions, and provides a small set of standard ISA options to extend the mandatory components. Profiles provide a convenient shorthand for describing the ISA portions of hardware and software platforms, and also guide the development of common software toolchains shared by different platforms that use the same profile. The intent is that the software ecosystem focus on supporting the profiles' mandatory base and standard options, instead of attempting to support every possible combination of individual extensions. Similarly, hardware vendors should aim to structure their offerings around standard profiles to increase the likelihood their designs will have mainstream software support.

*Profiles are not intended to prohibit the use of combinations of individual ISA extensions or the addition of custom extensions, which can continue to be used for more specialized applications albeit without the expectation of widespread software support or portability between hardware platforms.*

*As RISC-V evolves over time, the set of ISA features will grow, and new platforms will be added that may need different profiles. To manage this evolution, RISC-V is adopting a model of regular annual releases of new ISA profiles, following an ISA roadmap managed by the RISC-V Technical Steering Committee. The architecture profiles will also be used for branding and to advertise compatibility with the RISC-V standard.*

### 1.1. Profiles versus Platforms

Profiles only describe ISA features, not a complete execution environment.

A *software platform* is a specification for an execution environment, in which software targeted for that software platform can run.

A *hardware platform* is a specification for a hardware system (which can be viewed as a physical realization of an execution environment).

Both software and hardware platforms include specifications for many features beyond details of the ISA used by RISC-V harts in the platform (e.g., boot process, calling convention, behavior of environment calls, discovery mechanism, presence of certain memory-mapped hardware devices, etc.). Architecture profiles factor out ISA-specific definitions from platform definitions to allow ISA profiles to be reused across different platforms, and to be used by tools (e.g., compilers) that are common across many different platforms.

A platform can add additional constraints on top of those in a profile. For example, mandating an extension that is a standard option in the underlying profile, or constraining some implementation-specific parameter in the profile to lie within a certain range.

A platform cannot remove mandates or reduce other requirements in a profile.

*A new profile should be proposed if existing profiles do not match the needs of a new platform.*

## 1.2. Components of a Profile

### 1.2.1. Profile Family

Every profile is a member of a *profile family*. A profile family is a set of profiles that share the same base ISA but which vary in highest-supported privilege mode. The initial two types of family are:

- generic unprivileged instructions (I)
- application processors running rich operating systems (A)

*More profile families may be added over time.*

A profile family may be updated no more than annually, and the release calendar year is treated as part of the profile family name.

Each profile family is described in more detail below.

### 1.2.2. Profile Privilege Mode

RISC-V has a layered architecture supporting multiple privilege modes, and most RISC-V platforms support more than one privilege mode. Software is usually written assuming a particular privilege mode during execution. For example, application code is written assuming it will be run in user mode, and kernel code is written assuming it will be run in supervisor mode.

*Software can be run in a mode different than the one for which it was written. For example, privileged code using privileged ISA features can be run in a user-mode execution environment, but will then cause traps into the enclosing execution environment when privileged instructions are executed. This behavior might be exploited, for example, to emulate a privileged execution environment using a user-mode execution environment.*

The profile for a privilege mode describes the ISA features for an execution environment that has the eponymous privilege mode as the most-privileged mode available, but also includes all supported lower-privilege modes. In general, available instructions vary by privilege mode, and the behavior of RISC-V instructions can depend on the current privilege mode. For example, an S-mode profile includes U-mode as well as S-mode and describes the behavior of instructions when running in different modes in an S-mode execution environment, such as how an ecall instruction in U-mode causes a contained trap into an S-mode handler whereas an ecall in S-mode causes a requested trap out to the execution environment.

A profile may specify that certain conditions will cause a requested trap (such as an ecall made in the highest-supported privilege mode) or fatal trap to the enclosing execution environment. The profile does not specify the behavior of the enclosing execution environment iusually n handling requested or fatal traps.

*In particular, a profile does not specify the set of ECALLs available in the outer execution environment. This should be documented in the appropriate binary interface to the outer execution environment (e.g., Linux user ABI, or RISC-V SEE).*

*In general, a profile can be implemented by an execution environment using any hardware or software technique that provides compatible functionality, including pure software emulation.*

A profile does not specify any invisible traps.

*In particular, a profile does not constrain how invisible traps to a more-privileged mode can be used to emulate profile features.*

A more-privileged profile can always support running software to implement a less-privileged profile from the same profile family. For example, a platform supporting the S-mode profile can run a supervisor-mode operating system that provides user-mode execution environments supporting the U-mode profile.

*Instructions in a U-mode profile, which are all executed in user mode, have potentially different behaviors than instructions executed in user mode in an S-mode profile. For this reason, a U-mode profile cannot be considered a subset of an S-mode profile.*

### 1.2.3. Profile ISA Features

An architecture profile has a mandatory ratified base instruction set (RV32I or RV64I for the current profiles). The profile also includes ratified ISA extensions placed into two categories:

1. Mandatory
2. Optional

As the name implies, *Mandatory ISA extensions* are a required part of the profile. Implementations of the profile must provide these. The combination of the profile base ISA plus the mandatory ISA extensions are termed the profile *mandates*, and software using the profile can assume these always exist.

The *Optional* category (also known as *options*) contains extensions that may be added as options, and which are expected to be generally supported as options by the software ecosystem for this profile.

*The level of "support" for an Optional extension will likely vary greatly among different software components supporting a profile. Users*

> *would expect that software claiming compatibility with a profile would make use of any available supported options, but as a bare minimum software should not report errors or warnings when supported options are present in a system.*

An optional extension may comprise many individually named and ratified extensions but a profile option requires all constituent extensions are present. In particular, unless explicitly listed as a profile option, individual extensions are not by themselves a profile option even when required as part of a profile option. For example, the Zbkb extension is not by itself a profile option even though it is a required component of the Zkn option.

> *Profile optional extensions are intended to capture the granularity at which the broad software ecosystem is expected to cope with combinations of extensions.*

All components of a ratified profile must themselves have been ratified.

Platforms may provide a discovery mechanism to determine what optional extensions are present.

Extensions that are not explicitly listed in the mandatory or optional categories are termed *non-profile* extensions, and are not considered parts of the profile. Some non-profile extensions can be added to an implementation without conflicting with the mandatory or optional components of a profile. In this case, the implementation is still compatible with the profile even though additional non-profile extensions are present. Other non-profile extensions added to an implementation might alter or conflict with the behavior of the mandatory or optional extensions in a profile, in which case the implementation would not be compatible with the profile.

> *Extensions that are released after a given profile is released are by definition non-profile extensions. For example, mandatory or optional profile extensions for a new profile might be prototyped as non-profile extensions on an earlier profile.*

# 2. RVI Profile Class

The RVI profile class documents the initial set of unprivileged instructions.

## 2.1. RVI Description

The RVI profile class provides a generic target for software toolchains and represent the minimum level of compatibility with RISC-V ratified standards.

> *Profiles in this class are designated as unprivileged profiles as opposed to user-mode profiles. Code using this profile class can run in any privilege mode, and so requested and fatal traps may be horizontal traps into an execution environment running in the same privilege mode.*

## 2.2. RVI Naming Scheme

The profile class name is RVI (RISC-V base Integer instructions). A profile release name is an integer (currently 2 digits, could grow in the future). A full profile name is comprised of, in order:

- Prefix **RVI** for RISC-V Integer
- Profile release
- Privilege mode:
  - **U** Unprivileged (available to any privilege mode, **U** is **not** User-mode)
- A base ISA XLEN specifier (**32**, **64**)

> *Profile names are embeddable into RISC-V ISA naming strings. This implies that there will be no standard ISA extension with a name that matches the profile naming convention. This allows tools that process the RISC-V ISA naming string to parse and/or process a combined string.*

## 2.3. RVI Profile Releases

The following profile releases are defined in this profile class:

**Name**
: RVI20

**State**
: ratified

**Ratification date**
: 2023-04-03

# 3. RVI20 Profile Release

The two profiles RVI20U32 and RVI20U64 correspond to the RV32I and RV64I base ISAs respectively.

RVI20 has 13 associated implementation-defined parameters across all its defined profiles.

## 3.1. RVI20 Description

This profile release defines the RISC-V base ISA unprivileged instructions.

## 3.2. RVI20U32 Profile

This profile specifies the ISA features available to generic unprivileged execution environments.

### 3.2.1. Mandatory Extensions

The RVI20U32 Profile has 1 mandatory extension.

- **I** Base integer ISA (RV32I or RV64I)

  Version ~> 2.1

> *RVI is the mandatory base ISA for RVA, and is little-endian.*
>
> *As per the unprivileged architecture specification, the ecall instruction causes a requested trap to the execution environment.*
>
> *Misaligned loads and stores might not be supported.*
>
> *The fence.tso instruction is mandatory.*

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

### 3.2.2. Optional Extensions

The RVI20U32 Profile has 8 optional extensions.

#### Localized Options

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

> *The number of counters is platform-specific.*

#### Development Options

- **A** Atomic instructions

Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

**Expansion Options**

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

**Transitory Options**

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

### 3.2.3. Recommendations

Recommendations are not strictly mandated but are included to guide implementers making design choices.

- Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

### 3.2.4. Implementation-dependencies

RVI20U32 has 13 associated implementation-defined parameters.

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

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

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

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

**LRSC_RESERVATION_STRATEGY**
: Strategy used to handle reservation sets.

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

**MISALIGNED_AMO**
: whether or not the implementation supports misaligned atomics in main memory

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

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

**MUTABLE_MISA_D**

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

**HW_MSTATUS_FS_DIRTY_UPDATE**

Indicates whether or not hardware will write to mstatus.FS

Values are:

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

**MSTATUS_FS_LEGAL_VALUES**

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

**MUTABLE_MISA_F**

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

**MUTABLE_MISA_M**

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

**TIME_CSR_IMPLEMENTED**

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

Possible values:

**true**

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

**false**

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

## 3.3. RVI20U64 Profile

This profile specifies the ISA features available to generic unprivileged execution environments.

### 3.3.1. Mandatory Extensions

The RVI20U64 Profile has 1 mandatory extension.

- **I** Base integer ISA (RV32I or RV64I)

  Version ~> 2.1

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

### 3.3.2. Optional Extensions

The RVI20U64 Profile has 8 optional extensions.

#### Localized Options

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

**Development Options**

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

**Expansion Options**

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

**Transitory Options**

- **A** Atomic instructions

  Version = 2.1

- **C** Compressed instructions

  Version = 2.0

- **D** Double-precision floating-point

  Version = 2.2

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

- **F** Single-precision floating-point

  Version = 2.2

- **M** Integer multiply and divide instructions

  Version = 2.0

- **Zicntr** Architectural performance counters

  Version = 2.0

- **Zifencei** Instruction fence

  Version = 2.0

- **Zihpm** Programmable hardware performance counters

  Version = 2.0

  *The number of counters is platform-specific.*

### 3.3.3. Recommendations

Recommendations are not strictly mandated but are included to guide implementers making design choices.

- Implementations are strongly recommended to raise illegal-instruction exceptions on attempts to execute unimplemented opcodes.

### 3.3.4. Implementation-dependencies

RVI20U64 has 13 associated implementation-defined parameters.

**LRSC_FAIL_ON_NON_EXACT_LRSC**

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

**LRSC_FAIL_ON_VA_SYNONYM**

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

**LRSC_MISALIGNED_BEHAVIOR**

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

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

**LRSC_RESERVATION_STRATEGY**

Strategy used to handle reservation sets.

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

**MISALIGNED_AMO**

whether or not the implementation supports misaligned atomics in main memory

**MUTABLE_MISA_A**

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

**MUTABLE_MISA_C**

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

**MUTABLE_MISA_D**

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

**HW_MSTATUS_FS_DIRTY_UPDATE**

Indicates whether or not hardware will write to mstatus.FS

Values are:

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

**MSTATUS_FS_LEGAL_VALUES**

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

**MUTABLE_MISA_F**

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

**MUTABLE_MISA_M**

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

**TIME_CSR_IMPLEMENTED**

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

Possible values:

**true**

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

**false**

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

# Appendix A: Profile Comparisons

## A.1. Generic Unprivileged Profile Releases

The Generic Unprivileged processor kind has 1 processor profile releases that reference a total of 9 extensions.

*Table 1. Extension Presence*

| Extension | RVI20 |
|---|---|
| A | optional |
| C | optional |
| D | optional |
| F | optional |
| I | mandatory |
| M | optional |
| Zicntr | optional |
| Zifencei | optional |
| Zihpm | optional |

## A.2. RVI Profile Releases

The RVI Profile Class has 1 releases that reference a total of 9 extensions.

*Table 2. Extension Presence*

| Extension | RVI20 |
|---|---|
| A | optional |
| C | optional |
| D | optional |
| F | optional |
| I | mandatory |
| M | optional |
| Zicntr | optional |
| Zifencei | optional |
| Zihpm | optional |

## A.3. RVI20 Profiles

The RVI20 Profile Release has 2 profiles that reference a total of 9 extensions.

*Extensions present in a profile are also present in higher-privileged profiles in the same profile release.*

*Table 3. Extension Presence*

| Extension | RVI20U32 | RVI20U64 |
|---|---|---|
| A | optional | optional |
| C | optional | optional |
| D | optional | optional |
| F | optional | optional |
| I | mandatory | mandatory |
| M | optional | optional |
| Zicntr | optional | optional |
| Zifencei | optional | optional |
| Zihpm | optional | optional |

# Appendix B: Extension Details

## B.1. Extension A

**Long Name**: Atomic instructions
**Version Requirement**: = 2.1
A Extension Presence

| profile | v2.1.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.1.0**

**State**

ratified

**Ratification date**

2019-12

**Implies**

- Zaamo version 1.0.0
- Zalrsc version 1.0.0

### B.1.1. Synopsis

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

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

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

### B.1.2. Specifying Ordering of Atomic Instructions

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

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

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

### B.1.3. Instructions

The following 22 instructions are added by extension version 2.1.0 (the minimum version of this extension that satifies the extension requirement).

| | |
|---|---|
| amoadd.d | **Atomic fetch-and-add doubleword** |
| amoadd.w | **Atomic fetch-and-add word** |
| amoand.d | **Atomic fetch-and-and doubleword** |
| amoand.w | **Atomic fetch-and-and word** |
| amomax.d | **Atomic MAX doubleword** |
| amomax.w | **Atomic MAX word** |
| amomaxu.d | **Atomic MAX unsigned doubleword** |
| amomaxu.w | **Atomic MAX unsigned word** |
| amomin.d | **Atomic MIN doubleword** |
| amomin.w | **Atomic MIN word** |

| | |
|---|---|
| amominu.d | **Atomic MIN unsigned doubleword** |
| amominu.w | **Atomic MIN unsigned word** |
| amoor.d | **Atomic fetch-and-or doubleword** |
| amoor.w | **Atomic fetch-and-or word** |
| amoswap.d | **Atomic SWAP doubleword** |
| amoswap.w | **Atomic SWAP word** |
| amoxor.d | **Atomic fetch-and-xor doubleword** |
| amoxor.w | **Atomic fetch-and-xor word** |
| lr.d | **Load reserved doubleword** |
| lr.w | **Load reserved word** |
| sc.d | **Store conditional doubleword** |
| sc.w | **Store conditional word** |

### B.1.4. Parameters

This extension has the following parameters (AKA implementation options):

**LRSC_FAIL_ON_NON_EXACT_LRSC**

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

**LRSC_FAIL_ON_VA_SYNONYM**

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

**LRSC_MISALIGNED_BEHAVIOR**

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

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

**LRSC_RESERVATION_STRATEGY**

Strategy used to handle reservation sets.

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

**MISALIGNED_AMO**

whether or not the implementation supports misaligned atomics in main memory

**MUTABLE_MISA_A**

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

## B.2. Extension C

**Long Name**: Compressed instructions
**Version Requirement**: = 2.0
C Extension Presence

| profile | v2.0.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.0.0**

**State**

ratified

**Ratification date**

2019-12

**Implies**

- Zca version 1.0.0

- Zcf version 1.0.0
- Zcd version 1.0.0

### B.2.1. Synopsis

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

### B.2.2. Overview

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

*The initial RISC ISAs from the 1980s all picked performance over code size, which was reasonable for a workstation environment, but not for embedded systems. Hence, both ARM and MIPS subsequently made versions of the ISAs that offered smaller code size by offering an alternative 16-bit wide instruction set instead of the standard 32-bit wide instructions. The compressed RISC ISAs reduced*

*code size relative to their starting points by about 25-30%, yielding code that was significantly smaller than 80x86. This result surprised some, as their intuition was that the variable-length CISC ISA should be smaller than RISC ISAs that offered only 16-bit and 32-bit formats.*

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

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

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

### B.2.3. Compressed Instruction Formats

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

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

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

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

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

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

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

*Table 4. Compressed 16-bit RVC instruction formats*

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

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

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

#### B.2.4. Instructions

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

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

#### B.2.5. Parameters

This extension has the following parameters (AKA implementation options):

**MUTABLE_MISA_C**

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

### B.3. Extension D

**Long Name**: Double-precision floating-point
**Version Requirement**: = 2.2
D Extension Presence

| profile | v2.2.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.2.0**

**State**

ratified

**Ratification date**

2019-12

**Changes**

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

**Implies**

- F version 2.2.0

#### B.3.1. Synopsis

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

#### B.3.2. D Register State

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

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

### B.3.3. NaN Boxing of Narrower Values

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

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

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

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

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

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

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

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

### B.3.4. Instructions

The following 83 instructions are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

| | |
|---|---|
| fadd.d | **No synopsis available** |
| fclass.d | **No synopsis available** |
| fcvt.d.l | **No synopsis available** |
| fcvt.d.lu | **No synopsis available** |
| fcvt.d.s | **No synopsis available** |
| fcvt.d.w | **No synopsis available** |
| fcvt.d.wu | **No synopsis available** |
| fcvt.l.d | **No synopsis available** |
| fcvt.lu.d | **No synopsis available** |
| fcvt.s.d | **No synopsis available** |
| fcvt.w.d | **No synopsis available** |
| fcvt.wu.d | **No synopsis available** |
| fcvtmod.w.d | **No synopsis available** |
| fdiv.d | **No synopsis available** |
| feq.d | **No synopsis available** |
| fld | **No synopsis available** |
| fle.d | **No synopsis available** |
| fleq.d | **No synopsis available** |
| fli.d | **No synopsis available** |
| flt.d | **No synopsis available** |
| fltq.d | **No synopsis available** |

| | |
|---|---|
| fmadd.d | **No synopsis available** |
| fmax.d | **No synopsis available** |
| fmaxm.d | **No synopsis available** |
| fmin.d | **No synopsis available** |
| fminm.d | **No synopsis available** |
| fmsub.d | **No synopsis available** |
| fmul.d | **No synopsis available** |
| fmv.d.x | **No synopsis available** |
| fmv.x.d | **No synopsis available** |
| fmvh.x.d | **No synopsis available** |
| fmvp.d.x | **No synopsis available** |
| fnmadd.d | **No synopsis available** |
| fnmsub.d | **No synopsis available** |
| fround.d | **No synopsis available** |
| froundnx.d | **No synopsis available** |
| fsd | **No synopsis available** |
| fsgnj.d | **No synopsis available** |
| fsgnjn.d | **No synopsis available** |
| fsgnjx.d | **No synopsis available** |
| fsqrt.d | **No synopsis available** |
| fsub.d | **No synopsis available** |
| fadd.s | **Single-precision floating-point addition** |
| fclass.s | **Single-precision floating-point classify** |
| fcvt.l.s | **No synopsis available** |
| fcvt.lu.s | **No synopsis available** |
| fcvt.s.l | **No synopsis available** |
| fcvt.s.lu | **No synopsis available** |
| fcvt.s.w | **Convert signed 32-bit integer to single-precision float** |
| fcvt.s.wu | **Convert unsigned 32-bit integer to single-precision float** |
| fcvt.w.s | **Convert single-precision float to integer word to signed 32-bit integer** |
| fcvt.wu.s | **No synopsis available** |
| fdiv.s | **No synopsis available** |
| feq.s | **Single-precision floating-point equal** |
| fle.s | **Single-precision floating-point less than or equal** |
| flt.s | **Single-precision floating-point less than** |
| flw | **Single-precision floating-point load** |
| fmadd.s | **No synopsis available** |
| fmax.s | **No synopsis available** |
| fmin.s | **No synopsis available** |
| fmsub.s | **No synopsis available** |
| fmul.s | **No synopsis available** |
| fmv.w.x | **Single-precision floating-point move from integer** |
| fmv.x.w | **Move single-precision value from floating-point to integer register** |
| fnmadd.s | **No synopsis available** |
| fnmsub.s | **No synopsis available** |
| fsgnj.s | **Single-precision sign inject** |
| fsgnjn.s | **Single-precision sign inject negate** |
| fsgnjx.s | **Single-precision sign inject exclusive or** |
| fsqrt.s | **No synopsis available** |
| fsub.s | **Single-precision floating-point subtraction** |
| fsw | **Single-precision floating-point store** |
| c.fld | **Load double-precision** |
| c.fldsp | **Load doubleword into floating-point register from stack** |

| | |
|---|---|
| c.fsd | **Store double-precision** |
| c.fsdsp | **Store double-precision value to stack** |
| c.flw | **Load single-precision** |
| c.flwsp | **Load word into floating-point register from stack** |
| c.fsw | **Store single-precision** |
| c.fswsp | **Store single-precision value to stack** |
| fcvt.d.h | **No synopsis available** |
| fcvt.h.d | **No synopsis available** |
| fmv.h.x | **Half-precision floating-point move from integer** |

### B.3.5. CSRs

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

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

### B.3.6. Parameters

This extension has the following parameters (AKA implementation options):

**MUTABLE_MISA_D**

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

## B.4. Extension F

**Long Name**: Single-precision floating-point
**Version Requirement**: = 2.2
F Extension Presence

| profile | v2.2.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.2.0**

**State**

ratified

**Ratification date**

2019-12

**Changes**

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

### B.4.1. Synopsis

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

### B.4.2. F Register State

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

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

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

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

**Floating-Point Control and Status Register**

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

*Floating-point control and status register*

Unresolved directive in RVI20ProfileRelease.adoc - include::images/wavedrom/float-csr.adoc[]

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

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

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

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

*Table 7. Rounding mode encoding.*

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

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

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

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

*Table 8. Accrued exception flag encoding.*

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

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

### B.4.3. NaN Generation and Propagation

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

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

*Implementers are free to provide a NaN payload propagation scheme as a nonstandard extension enabled by a nonstandard operating mode. However, the canonical NaN scheme described above must always be supported and should be the default mode.*

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

### B.4.4. Subnormal Arithmetic

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

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

*Detecting tininess after rounding results in fewer spurious underflow signals.*

### B.4.5. Instructions

The following 35 instructions are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

| | |
|---|---|
| fadd.s | **Single-precision floating-point addition** |
| fclass.s | **Single-precision floating-point classify** |
| fcvt.l.s | **No synopsis available** |

| | |
|---|---|
| fcvt.lu.s | **No synopsis available** |
| fcvt.s.l | **No synopsis available** |
| fcvt.s.lu | **No synopsis available** |
| fcvt.s.w | **Convert signed 32-bit integer to single-precision float** |
| fcvt.s.wu | **Convert unsigned 32-bit integer to single-precision float** |
| fcvt.w.s | **Convert single-precision float to integer word to signed 32-bit integer** |
| fcvt.wu.s | **No synopsis available** |
| fdiv.s | **No synopsis available** |
| feq.s | **Single-precision floating-point equal** |
| fle.s | **Single-precision floating-point less than or equal** |
| flt.s | **Single-precision floating-point less than** |
| flw | **Single-precision floating-point load** |
| fmadd.s | **No synopsis available** |
| fmax.s | **No synopsis available** |
| fmin.s | **No synopsis available** |
| fmsub.s | **No synopsis available** |
| fmul.s | **No synopsis available** |
| fmv.w.x | **Single-precision floating-point move from integer** |
| fmv.x.w | **Move single-precision value from floating-point to integer register** |
| fnmadd.s | **No synopsis available** |
| fnmsub.s | **No synopsis available** |
| fsgnj.s | **Single-precision sign inject** |
| fsgnjn.s | **Single-precision sign inject negate** |
| fsgnjx.s | **Single-precision sign inject exclusive or** |
| fsqrt.s | **No synopsis available** |
| fsub.s | **Single-precision floating-point subtraction** |
| fsw | **Single-precision floating-point store** |
| c.flw | **Load single-precision** |
| c.flwsp | **Load word into floating-point register from stack** |
| c.fsw | **Store single-precision** |
| c.fswsp | **Store single-precision value to stack** |
| fmv.h.x | **Half-precision floating-point move from integer** |

### B.4.6. CSRs

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

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

### B.4.7. Parameters

This extension has the following parameters (AKA implementation options):

**HW_MSTATUS_FS_DIRTY_UPDATE**

Indicates whether or not hardware will write to mstatus.FS

Values are:

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

**MSTATUS_FS_LEGAL_VALUES**

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

**MUTABLE_MISA_F**

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

## B.5. Extension I

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

| profile | v2.1.0 |
|---|---|
| RVI20U32 | mandatory |
| RVI20U64 | mandatory |

**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

### B.5.1. Synopsis

Base integer instructions — TODO

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

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

*Misaligned loads and stores might not be supported.*

*The fence.tso instruction is mandatory.*

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

### B.5.2. Instructions

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

| | |
|---|---|
| add | **Integer add** |
| addi | **Add immediate** |
| addiw | **Add immediate word** |
| addw | **Add word** |
| and | **And** |
| andi | **And immediate** |
| auipc | **Add upper immediate to pc** |
| beq | **Branch if equal** |
| bge | **Branch if greater than or equal** |
| bgeu | **Branch if greater than or equal unsigned** |
| blt | **Branch if less than** |
| bltu | **Branch if less than unsigned** |
| bne | **Branch if not equal** |
| ebreak | **Breakpoint exception** |
| ecall | **Environment call** |
| fence.tso | **Memory ordering fence, total store ordering** |
| fence | **Memory oxdering fence** |
| jal | **Jump and link** |
| jalr | **Jump and link register** |
| lb | **Load byte** |

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

## B.6. Extension M

**Long Name**: Integer multiply and divide instructions
**Version Requirement**: = 2.0
M Extension Presence

| profile | v2.0.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.0.0**

**State**

ratified

**Ratification date**

2019-12

### B.6.1. Synopsis

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

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

### B.6.2. Instructions

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

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

### B.6.3. Parameters

This extension has the following parameters (AKA implementation options):

**MUTABLE_MISA_M**

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

## B.7. Extension Zicntr

**Long Name**: Architectural performance counters
**Version Requirement**: = 2.0
Zicntr Extension Presence

| profile | v2.0.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.0.0**

**State**

ratified

**Ratification date**

2019-12

### B.7.1. Synopsis

Architectural performance counters

### B.7.2. CSRs

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

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

### B.7.3. Parameters

This extension has the following parameters (AKA implementation options):

**TIME_CSR_IMPLEMENTED**

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

Possible values:

**true**

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

**false**

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

## B.8. Extension Zifencei

**Long Name**: Instruction fence
**Version Requirement**: = 2.0
Zifencei Extension Presence

| profile | v2.0.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.0.0**

**State**

ratified

### B.8.1. Synopsis

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

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

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

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

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

*Second, the instruction is not powerful enough to make available at user level in a Unix-like operating system environment. The FENCE.I only synchronizes the local hart, and the OS can reschedule the user hart to a different physical hart after the FENCE.I. This would require the OS to execute an additional FENCE.I as part of every context migration. For this reason, the standard Linux ABI has removed FENCE.I from user-level and now requires a system call to maintain instruction-fetch coherence, which allows the OS to minimize the number of FENCE.I executions required on current systems and provides forward-compatibility with future improved instruction-fetch coherence mechanisms.*

*Future approaches to instruction-fetch coherence under discussion include providing more restricted versions of FENCE.I that only target a given address specified in rs1, and/or allowing software to use an ABI that relies on machine-mode cache-maintenance operations.*

### B.8.2. Instructions

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

| | |
|---|---|
| fence.i | **Instruction fence** |

## B.9. Extension Zihpm

**Long Name**: Programmable hardware performance counters
**Version Requirement**: = 2.0
Zihpm Extension Presence

| profile | v2.0.0 |
|---|---|
| RVI20U32 | optional |
| RVI20U64 | optional |

**2.0.0**

**State**

ratified

### B.9.1. Synopsis

Programmable hardware performance counters

*The number of counters is platform-specific.*

### B.9.2. CSRs

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

| Name | Long Name | Address | Mode |
|---|---|---|---|
| hpmcounter10 | User-mode Hardware Performance Counter 7 | 0xc0a | U |
| hpmcounter11 | User-mode Hardware Performance Counter 8 | 0xc0b | U |
| hpmcounter12 | User-mode Hardware Performance Counter 9 | 0xc0c | U |
| hpmcounter13 | User-mode Hardware Performance Counter 10 | 0xc0d | U |
| hpmcounter14 | User-mode Hardware Performance Counter 11 | 0xc0e | U |
| hpmcounter15 | User-mode Hardware Performance Counter 12 | 0xc0f | U |
| hpmcounter16 | User-mode Hardware Performance Counter 13 | 0xc10 | U |
| hpmcounter17 | User-mode Hardware Performance Counter 14 | 0xc11 | U |
| hpmcounter18 | User-mode Hardware Performance Counter 15 | 0xc12 | U |
| hpmcounter19 | User-mode Hardware Performance Counter 16 | 0xc13 | U |
| hpmcounter20 | User-mode Hardware Performance Counter 17 | 0xc14 | U |
| hpmcounter21 | User-mode Hardware Performance Counter 18 | 0xc15 | U |
| hpmcounter22 | User-mode Hardware Performance Counter 19 | 0xc16 | U |
| hpmcounter23 | User-mode Hardware Performance Counter 20 | 0xc17 | U |
| hpmcounter24 | User-mode Hardware Performance Counter 21 | 0xc18 | U |
| hpmcounter25 | User-mode Hardware Performance Counter 22 | 0xc19 | U |
| hpmcounter26 | User-mode Hardware Performance Counter 23 | 0xc1a | U |
| hpmcounter27 | User-mode Hardware Performance Counter 24 | 0xc1b | U |
| hpmcounter28 | User-mode Hardware Performance Counter 25 | 0xc1c | U |
| hpmcounter29 | User-mode Hardware Performance Counter 26 | 0xc1d | U |
| hpmcounter3 | User-mode Hardware Performance Counter 0 | 0xc03 | U |
| hpmcounter30 | User-mode Hardware Performance Counter 27 | 0xc1e | U |
| hpmcounter31 | User-mode Hardware Performance Counter 28 | 0xc1f | U |
| hpmcounter4 | User-mode Hardware Performance Counter 1 | 0xc04 | U |
| hpmcounter5 | User-mode Hardware Performance Counter 2 | 0xc05 | U |
| hpmcounter6 | User-mode Hardware Performance Counter 3 | 0xc06 | U |
| hpmcounter7 | User-mode Hardware Performance Counter 4 | 0xc07 | U |
| hpmcounter8 | User-mode Hardware Performance Counter 5 | 0xc08 | U |
| hpmcounter9 | User-mode Hardware Performance Counter 6 | 0xc09 | U |

# Appendix C: Instruction Details

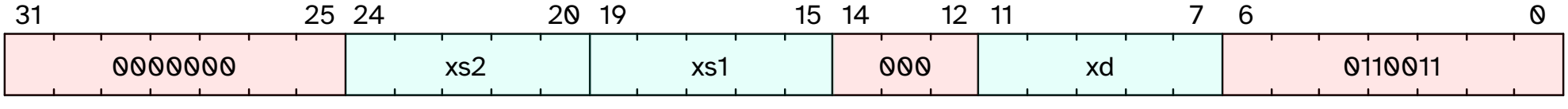
## C.1. add

### Integer add

This instruction is defined by:

- I, version >= I@2.1.0

### C.1.1. Encoding




### C.1.2. Description

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

### C.1.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.1.4. Decode Variables

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

### C.1.5. IDL Operation

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

### C.1.6. Sail Operation

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

### C.1.7. Exceptions

This instruction does not generate synchronous exceptions.

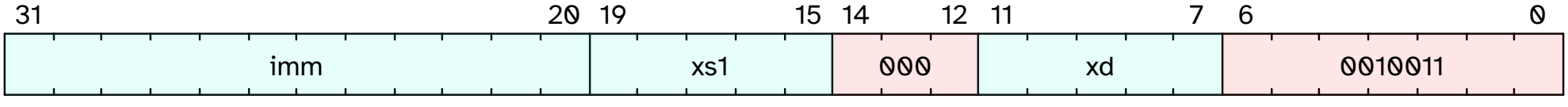
## C.2. addi

### Add immediate

This instruction is defined by:

- I, version >= I@2.1.0

### C.2.1. Encoding




### C.2.2. Description

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

### C.2.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.2.4. Decode Variables

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

### C.2.5. IDL Operation

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

### C.2.6. Sail Operation

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

### C.2.7. Exceptions

This instruction does not generate synchronous exceptions.

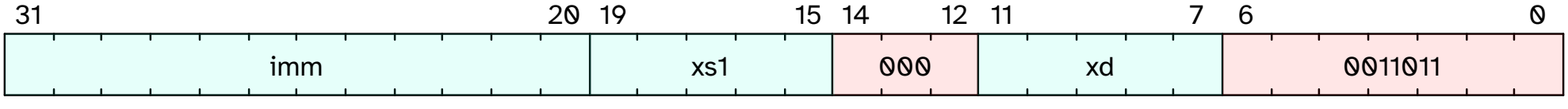
## C.3. addiw

**Add immediate word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.3.1. Encoding




### C.3.2. Description

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

### C.3.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.3.4. Decode Variables

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

### C.3.5. IDL Operation

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

### C.3.6. Sail Operation

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

### C.3.7. Exceptions

This instruction does not generate synchronous exceptions.

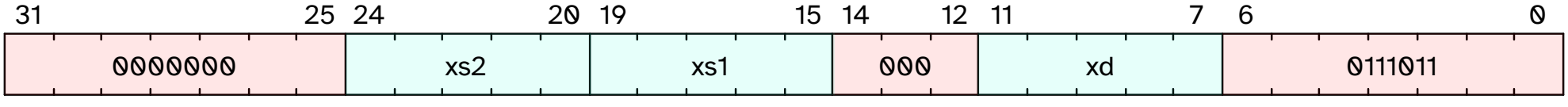
## C.4. addw

**Add word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.4.1. Encoding




### C.4.2. Description

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

### C.4.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.4.4. Decode Variables

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

### C.4.5. IDL Operation

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

### C.4.6. Sail Operation

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

### C.4.7. Exceptions

This instruction does not generate synchronous exceptions.

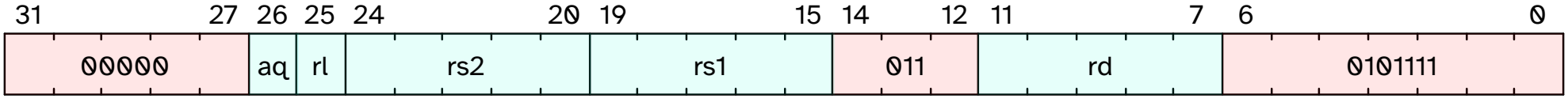
## C.5. amoadd.d

**Atomic fetch-and-add doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.5.1. Encoding




### C.5.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Add the value of register *rs2* to the loaded value
- Write the sum to the address in *rs1*

### C.5.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.5.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.5.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Add, aq, rl, $encoding);
```

### C.5.6. Sail Operation

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

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

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

### C.5.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

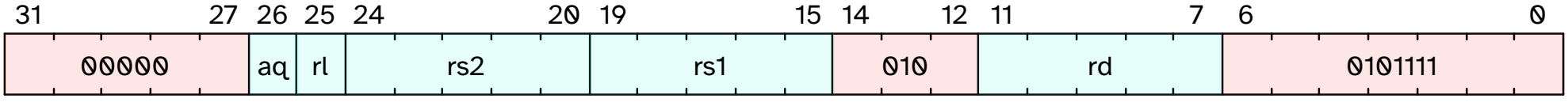
## C.6. amoadd.w

**Atomic fetch-and-add word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.6.1. Encoding




### C.6.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Add the least-significant word of register *rs2* to the loaded value
- Write the sum to the address in *rs1*

### C.6.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.6.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.6.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Add, aq, rl, $encoding);
```

### C.6.6. Sail Operation

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

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

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

### C.6.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

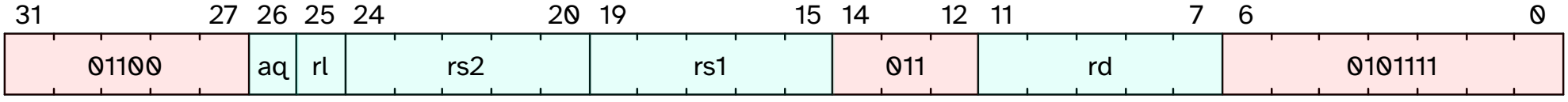
## C.7. amoand.d

**Atomic fetch-and-and doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.7.1. Encoding




### C.7.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- AND the value of register *rs2* to the loaded value
- Write the result to the address in *rs1*

### C.7.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.7.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.7.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::And, aq, rl, $encoding);
```

### C.7.6. Sail Operation

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

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

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

### C.7.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

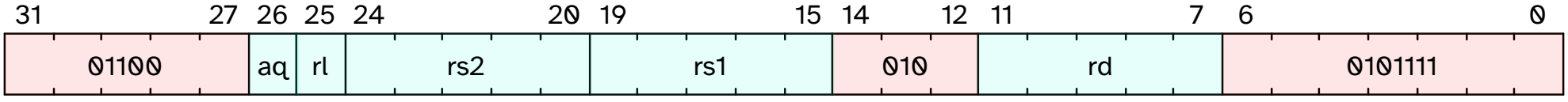
## C.8. amoand.w

**Atomic fetch-and-and word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.8.1. Encoding




### C.8.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- AND the least-significant word of register *rs2* to the loaded value
- Write the result to the address in *rs1*

### C.8.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.8.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.8.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::And, aq, rl, $encoding);
```

### C.8.6. Sail Operation

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

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

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

### C.8.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

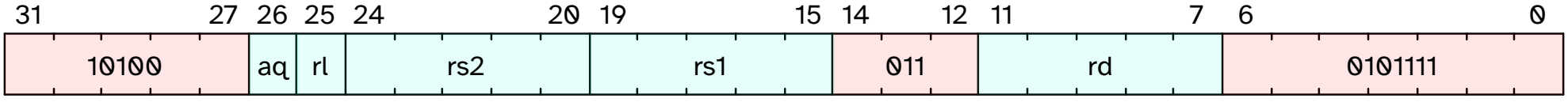
## C.9. amomax.d

**Atomic MAX doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.9.1. Encoding




### C.9.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Signed compare the value of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

### C.9.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.9.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.9.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Max, aq, rl, $encoding);
```

### C.9.6. Sail Operation

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

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

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

### C.9.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

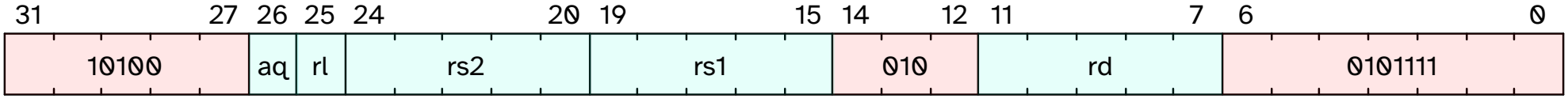
## C.10. amomax.w

**Atomic MAX word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.10.1. Encoding




### C.10.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Signed compare the least-significant word of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

### C.10.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.10.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.10.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Max, aq, rl, $encoding);
```

### C.10.6. Sail Operation

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

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

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

### C.10.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

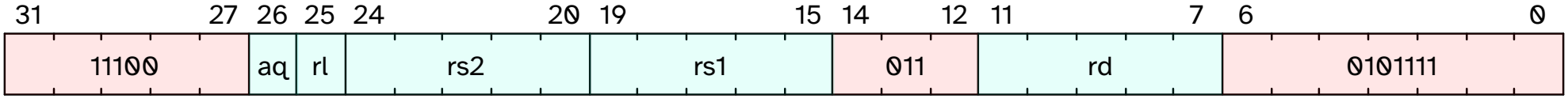
## C.11. amomaxu.d

**Atomic MAX unsigned doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.11.1. Encoding




### C.11.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Unsigned compare the value of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

### C.11.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.11.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.11.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Maxu, aq, rl, $encoding);
```

### C.11.6. Sail Operation

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

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

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

### C.11.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

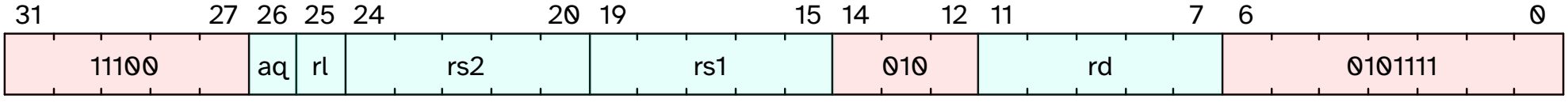
## C.12. amomaxu.w

**Atomic MAX unsigned word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.12.1. Encoding




### C.12.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Unsigned compare the least-significant word of register *rs2* to the loaded value, and select the maximum value
- Write the maximum to the address in *rs1*

### C.12.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.12.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.12.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Maxu, aq, rl, $encoding);
```

### C.12.6. Sail Operation

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

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

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

### C.12.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

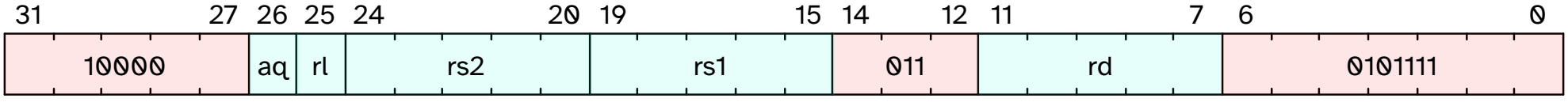
## C.13. amomin.d

**Atomic MIN doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.13.1. Encoding




### C.13.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Signed compare the value of register *rs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *rs1*

### C.13.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.13.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.13.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Min, aq, rl, $encoding);
```

### C.13.6. Sail Operation

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

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

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

### C.13.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

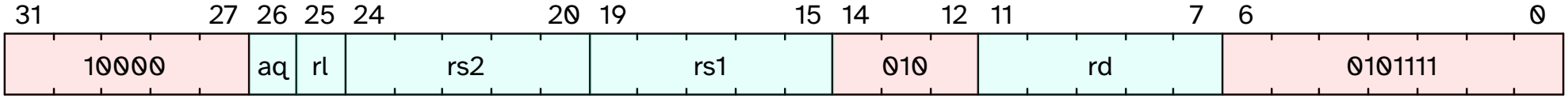
## C.14. amomin.w

**Atomic MIN word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.14.1. Encoding




### C.14.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Signed compare the least-significant word of register *rs2* to the loaded value, and select the minimum value
- Write the result to the address in *rs1*

### C.14.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.14.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.14.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Min, aq, rl, $encoding);
```

### C.14.6. Sail Operation

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

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

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

### C.14.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

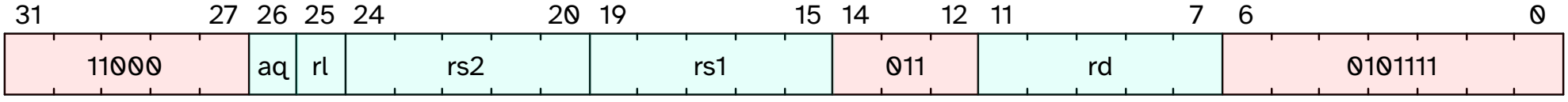
## C.15. amominu.d

**Atomic MIN unsigned doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.15.1. Encoding




### C.15.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- Unsigned compare the value of register *rs2* to the loaded value, and select the minimum value
- Write the minimum to the address in *rs1*

### C.15.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.15.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.15.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Minu, aq, rl, $encoding);
```

### C.15.6. Sail Operation

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

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

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

### C.15.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

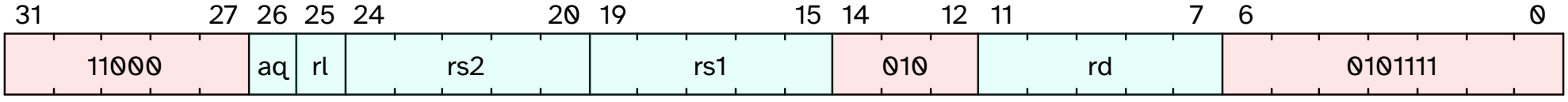
## C.16. amominu.w

**Atomic MIN unsigned word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.16.1. Encoding




### C.16.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Unsigned compare the least-significant word of register *rs2* to the loaded word, and select the minimum value
- Write the result to the address in *rs1*

### C.16.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.16.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.16.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Minu, aq, rl, $encoding);
```

### C.16.6. Sail Operation

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

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

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

### C.16.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

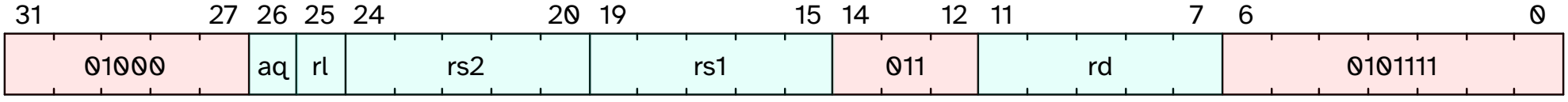
## C.17. amoor.d

**Atomic fetch-and-or doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.17.1. Encoding




### C.17.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- OR the value of register *rs2* to the loaded value
- Write the result to the address in *rs1*

### C.17.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.17.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.17.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Or, aq, rl, $encoding);
```

### C.17.6. Sail Operation

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

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

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

### C.17.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

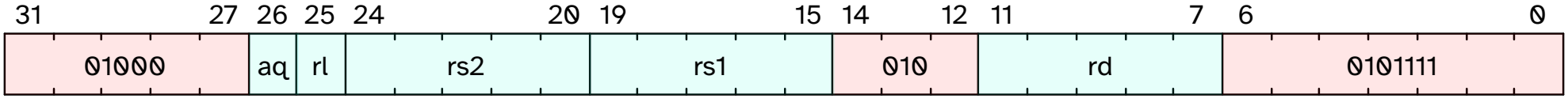
## C.18. amoor.w

**Atomic fetch-and-or word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.18.1. Encoding




### C.18.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- OR the least-significant word of register *rs2* to the loaded value
- Write the result to the address in *rs1*

### C.18.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.18.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.18.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Or, aq, rl, $encoding);
```

### C.18.6. Sail Operation

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

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

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

### C.18.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

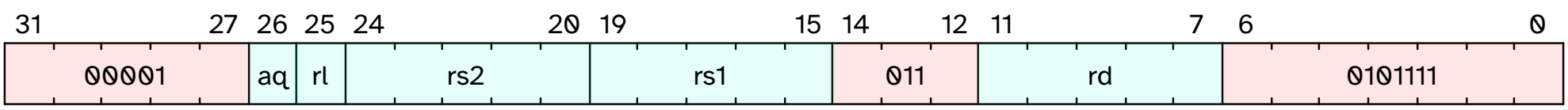
## C.19. amoswap.d

**Atomic SWAP doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.19.1. Encoding




### C.19.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the value into *rd*
- Store the value of register *rs2* to the address in *rs1*

### C.19.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.19.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.19.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Swap, aq, rl, $encoding);
```

### C.19.6. Sail Operation

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

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

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

### C.19.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

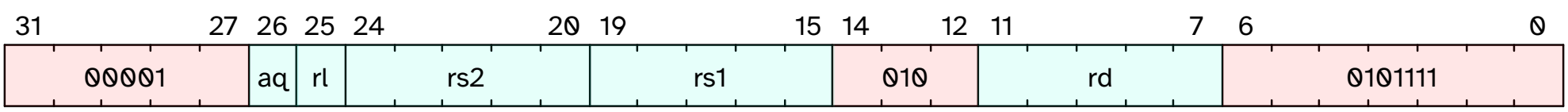
## C.20. amoswap.w

**Atomic SWAP word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.20.1. Encoding




### C.20.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- Store the least-significant word of register *rs2* to the address in *rs1*

### C.20.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.20.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.20.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Swap, aq, rl, $encoding);
```

### C.20.6. Sail Operation

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

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

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

### C.20.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

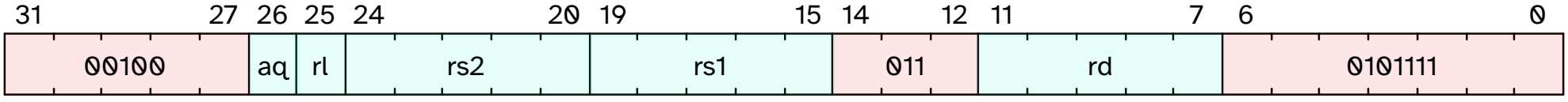
## C.21. amoxor.d

**Atomic fetch-and-xor doubleword**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.21.1. Encoding




### C.21.2. Description

Atomically:

- Load the doubleword at address *rs1*
- Write the loaded value into *rd*
- XOR the value of register *rs2* to the loaded value
- Write the result to the address in *rs1*

### C.21.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.21.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.21.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<64>(virtual_address, X[rs2], AmoOperation::Xor, aq, rl, $encoding);
```

### C.21.6. Sail Operation

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

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

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

### C.21.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

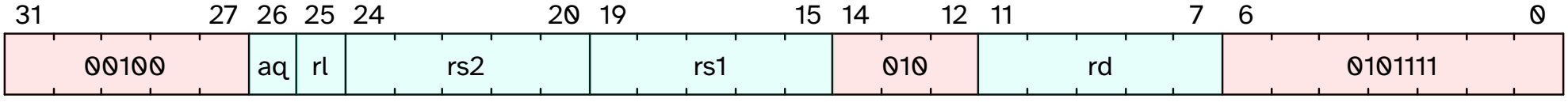
## C.22. amoxor.w

**Atomic fetch-and-xor word**

This instruction is defined by:

- Zaamo, version >= Zaamo@1.0.0

### C.22.1. Encoding




### C.22.2. Description

Atomically:

- Load the word at address *rs1*
- Write the sign-extended value into *rd*
- XOR the least-significant word of register *rs2* to the loaded value
- Write the result to the address in *rs1*

### C.22.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.22.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.22.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
X[rd] = amo<32>(virtual_address, X[rs2][31:0], AmoOperation::Xor, aq, rl, $encoding);
```

### C.22.6. Sail Operation

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

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

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

### C.22.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

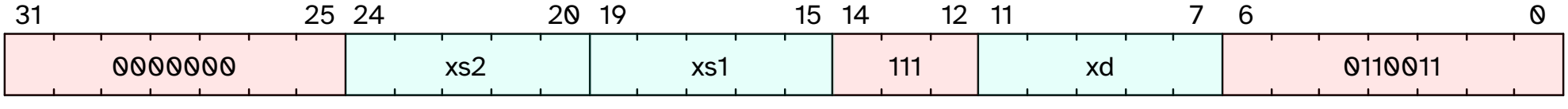
## C.23. and

**And**

This instruction is defined by:

- I, version >= I@2.1.0

### C.23.1. Encoding




### C.23.2. Description

And xs1 with xs2, and store the result in xd

### C.23.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.23.4. Decode Variables

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

### C.23.5. IDL Operation

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

### C.23.6. Sail Operation

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

### C.23.7. Exceptions

This instruction does not generate synchronous exceptions.

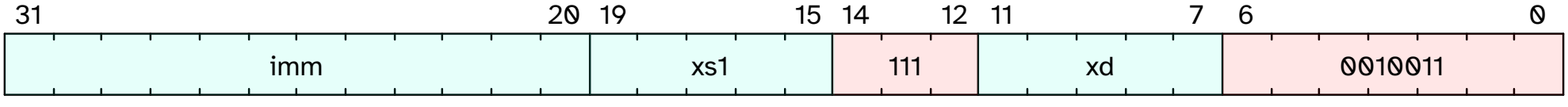
## C.24. andi

**And immediate**

This instruction is defined by:

- I, version >= I@2.1.0

### C.24.1. Encoding




### C.24.2. Description

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

### C.24.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.24.4. Decode Variables

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

### C.24.5. IDL Operation

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

### C.24.6. Sail Operation

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

### C.24.7. Exceptions

This instruction does not generate synchronous exceptions.

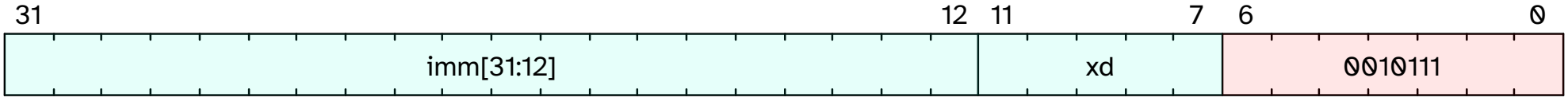
## C.25. auipc

**Add upper immediate to pc**

This instruction is defined by:

- I, version >= I@2.1.0

### C.25.1. Encoding




### C.25.2. Description

Add an immediate to the current PC.

### C.25.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.25.4. Decode Variables

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

### C.25.5. IDL Operation

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

### C.25.6. Sail Operation

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

### C.25.7. Exceptions

This instruction does not generate synchronous exceptions.

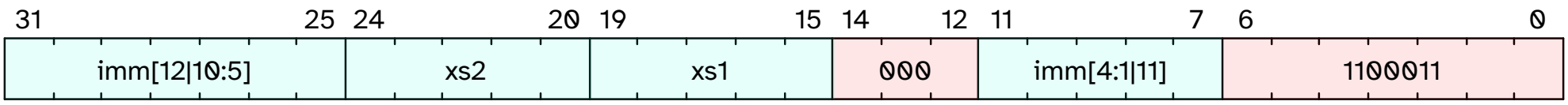
## C.26. beq

**Branch if equal**

This instruction is defined by:

- I, version >= I@2.1.0

### C.26.1. Encoding




### C.26.2. Description

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

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

### C.26.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.26.4. Decode Variables

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

### C.26.5. IDL Operation

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

### C.26.6. Sail Operation

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

```
      }
    }
  } else RETIRE_SUCCESS
}
```

### C.26.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

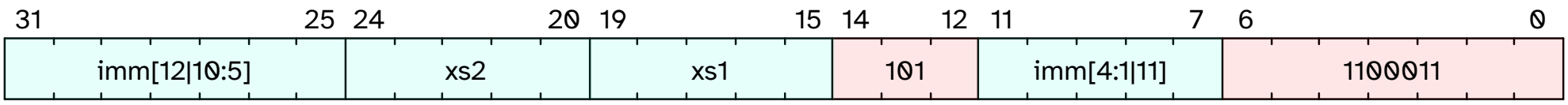
## C.27. bge

**Branch if greater than or equal**

This instruction is defined by:

- I, version >= I@2.1.0

### C.27.1. Encoding




### C.27.2. Description

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

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

### C.27.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.27.4. Decode Variables

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

### C.27.5. IDL Operation

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

### C.27.6. Sail Operation

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

```
      }
    }
  } else RETIRE_SUCCESS
}
```

### C.27.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

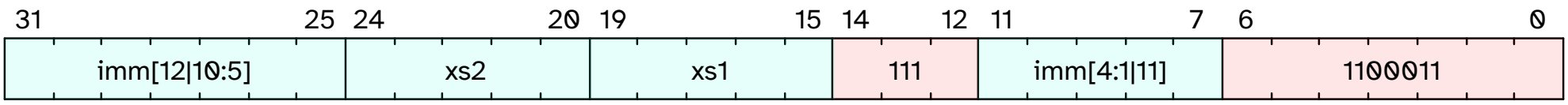
## C.28. bgeu

**Branch if greater than or equal unsigned**

This instruction is defined by:

- I, version >= I@2.1.0

### C.28.1. Encoding




### C.28.2. Description

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

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

### C.28.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.28.4. Decode Variables

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

### C.28.5. IDL Operation

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

### C.28.6. Sail Operation

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

```
      }
    }
  } else RETIRE_SUCCESS
}
```

### C.28.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

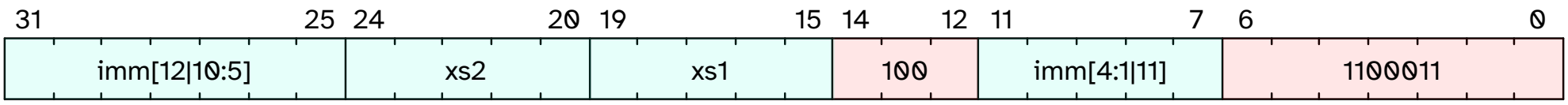
## C.29. blt

**Branch if less than**

This instruction is defined by:

- I, version >= I@2.1.0

### C.29.1. Encoding




### C.29.2. Description

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

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

### C.29.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.29.4. Decode Variables

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

### C.29.5. IDL Operation

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

### C.29.6. Sail Operation

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

```
      }
    }
  } else RETIRE_SUCCESS
}
```

### C.29.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

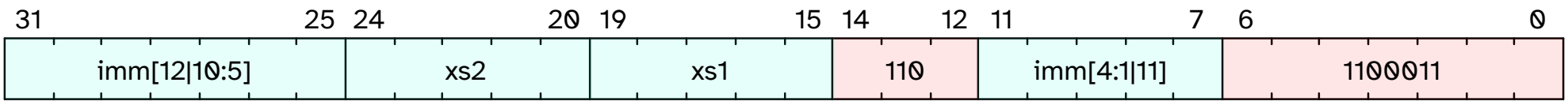
## C.30. bltu

**Branch if less than unsigned**

This instruction is defined by:

- I, version >= I@2.1.0

### C.30.1. Encoding




### C.30.2. Description

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

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

### C.30.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.30.4. Decode Variables

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

### C.30.5. IDL Operation

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

### C.30.6. Sail Operation

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

```
      }
    }
  } else RETIRE_SUCCESS
}
```

### C.30.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

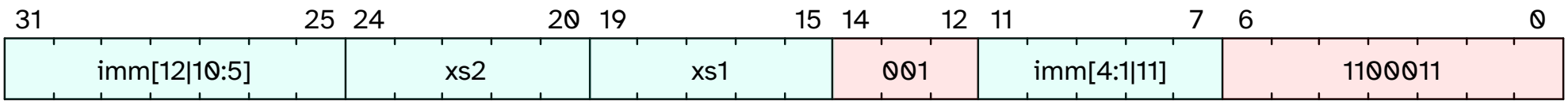
## C.31. bne

**Branch if not equal**

This instruction is defined by:

- I, version >= I@2.1.0

### C.31.1. Encoding




### C.31.2. Description

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

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

### C.31.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.31.4. Decode Variables

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

### C.31.5. IDL Operation

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

### C.31.6. Sail Operation

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

```
      }
    }
  } else RETIRE_SUCCESS
}
```

### C.31.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

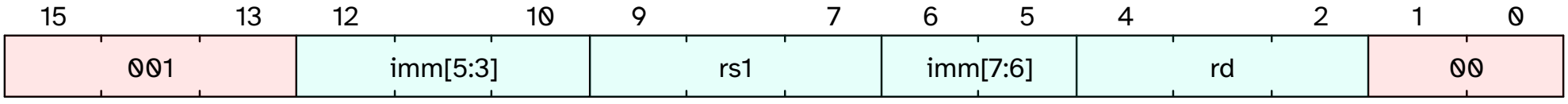
## C.32. c.fld

### Load double-precision

This instruction is defined by:

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

### C.32.1. Encoding




### C.32.2. Description

Loads a double precision floating-point 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 fld `rd, offset(rs1)`.

### C.32.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.32.4. 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];
```

### C.32.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(rs1)] + imm;
X[creg2reg(rd)] = sext(read_memory<64>(virtual_address, $encoding), 64);
```

### C.32.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

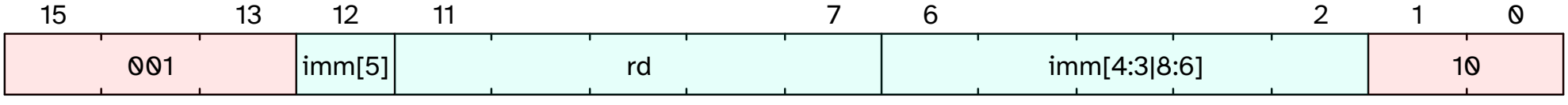
## C.33. c.fldsp

**Load doubleword into floating-point register from stack**

This instruction is defined by:

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

### C.33.1. Encoding




### C.33.2. Description

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

### C.33.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.33.4. Decode Variables

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

### C.33.5. IDL Operation

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

### C.33.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

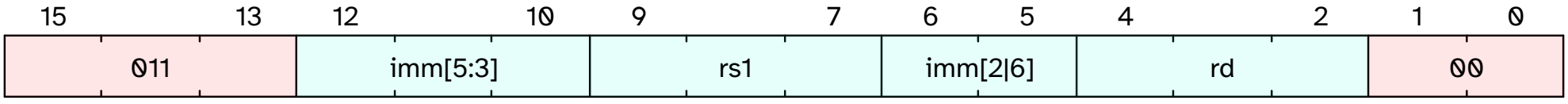
## C.34. c.flw

### Load single-precision

This instruction is defined by:

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

### C.34.1. Encoding




### C.34.2. Description

Loads a single precision floating-point 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 flw `rd, offset(rs1)`.

### C.34.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.34.4. 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];
```

### C.34.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(rs1)] + imm;
X[creg2reg(rd)] = sext(read_memory<32>(virtual_address, $encoding), 32);
```

### C.34.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

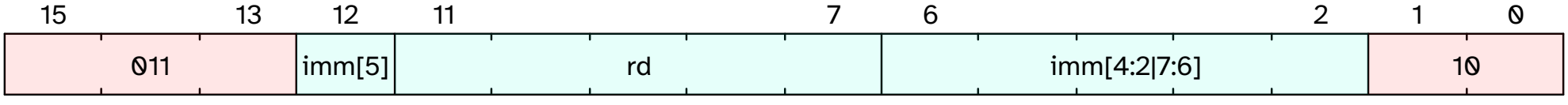
## C.35. c.flwsp

**Load word into floating-point register from stack**

This instruction is defined by:

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

### C.35.1. Encoding




### C.35.2. Description

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

### C.35.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.35.4. Decode Variables

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

### C.35.5. IDL Operation

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

### C.35.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

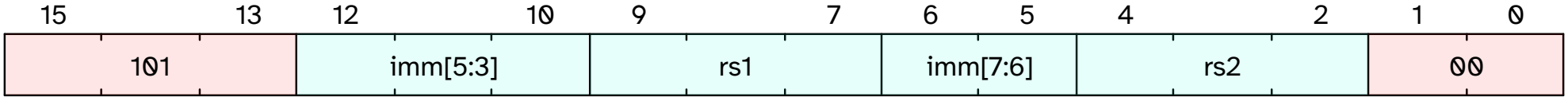
## C.36. c.fsd

**Store double-precision**

This instruction is defined by:

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

### C.36.1. Encoding




### C.36.2. Description

Stores a double precision floating-point 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 fsd `rs2, offset(rs1)`.

### C.36.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.36.4. 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];
```

### C.36.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(rs1)] + imm;
write_memory<64>(virtual_address, X[creg2reg(rs2)], $encoding);
```

### C.36.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

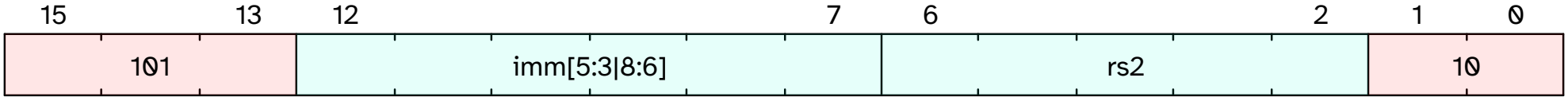
## C.37. c.fsdsp

**Store double-precision value to stack**

This instruction is defined by:

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

### C.37.1. Encoding




### C.37.2. Description

Stores a double-precision floating-point value in floating-point 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 fsd `rs2, offset(x2)`.

### C.37.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.37.4. Decode Variables

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

### C.37.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (implemented?(ExtensionName::D) && (misa.D == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<64>(virtual_address, f[rs2][63:0], $encoding);
```

### C.37.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

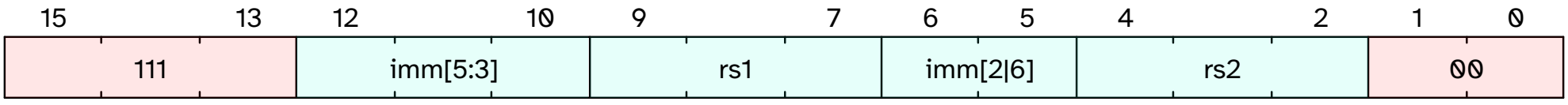
## C.38. c.fsw

**Store single-precision**

This instruction is defined by:

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

### C.38.1. Encoding




### C.38.2. Description

Stores a single precision floating-point 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 fsw `rs2, offset(rs1)`.

### C.38.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.38.4. 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];
```

### C.38.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (misa.C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(rs1)] + imm;
write_memory<32>(virtual_address, X[creg2reg(rs2)][31:0], $encoding);
```

### C.38.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

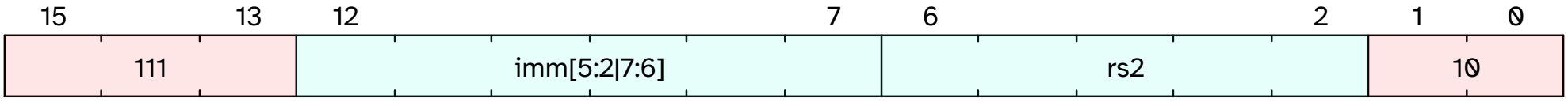
## C.39. c.fswsp

**Store single-precision value to stack**

This instruction is defined by:

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

### C.39.1. Encoding




### C.39.2. Description

Stores a single-precision floating-point value in floating-point 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 fsw `rs2, offset(x2)`.

### C.39.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.39.4. Decode Variables

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

### C.39.5. IDL Operation

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

### C.39.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

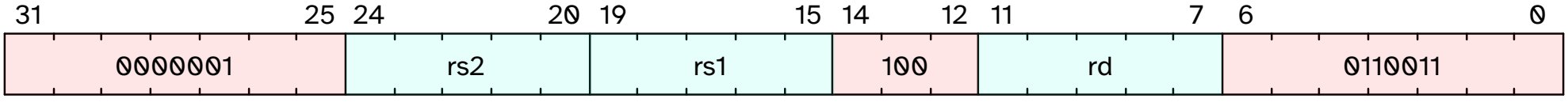
## C.40. div

### Signed division

This instruction is defined by:

- M, version >= M@2.0.0

### C.40.1. Encoding




### C.40.2. Description

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

Division by zero will put -1 into rd.

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

### C.40.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.40.4. Decode Variables

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

### C.40.5. IDL Operation

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

### C.40.6. Sail Operation

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

### C.40.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

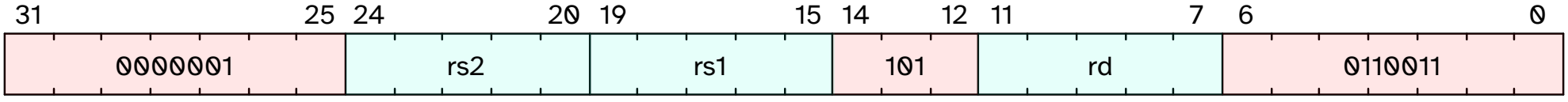
## C.41. divu

### Unsigned division

This instruction is defined by:

- M, version >= M@2.0.0

### C.41.1. Encoding




### C.41.2. Description

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

The remainder is discarded.

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

### C.41.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.41.4. Decode Variables

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

### C.41.5. IDL Operation

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

### C.41.6. Sail Operation

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

### C.41.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

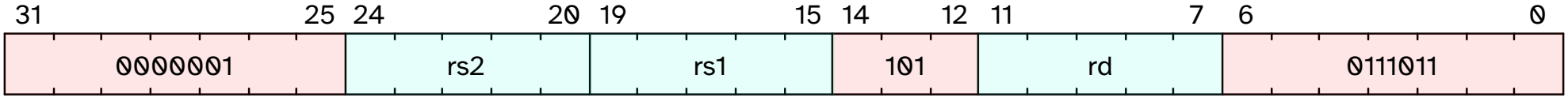
## C.42. divuw

### Unsigned 32-bit division

This instruction is defined by:

- M, version >= M@2.0.0

### C.42.1. Encoding




### C.42.2. Description

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

The remainder is discarded.

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

### C.42.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.42.4. Decode Variables

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

### C.42.5. IDL Operation

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

### C.42.6. Sail Operation

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

### C.42.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

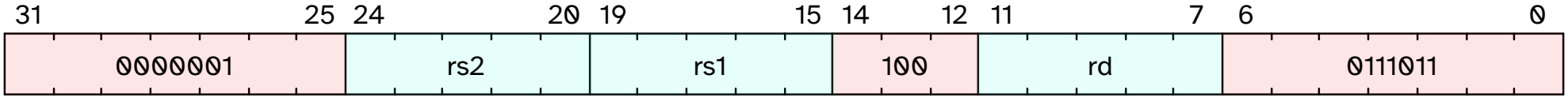
## C.43. divw

**Signed 32-bit division**

This instruction is defined by:

- M, version >= M@2.0.0

### C.43.1. Encoding




### C.43.2. Description

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

The remainder is discarded.

Division by zero will put -1 into rd.

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

### C.43.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.43.4. Decode Variables

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

### C.43.5. IDL Operation

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

### C.43.6. Sail Operation

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

```
    }
}
```

### C.43.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

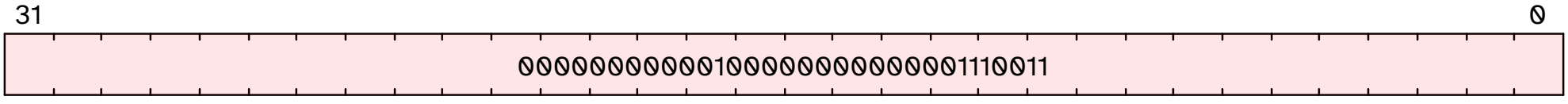
## C.44. ebreak

**Breakpoint exception**

This instruction is defined by:

- I, version >= I@2.1.0

### C.44.1. Encoding




### C.44.2. Description

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

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

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

### C.44.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.44.4. Decode Variables

```
```

### C.44.5. IDL Operation

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

### C.44.6. Sail Operation

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

### C.44.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

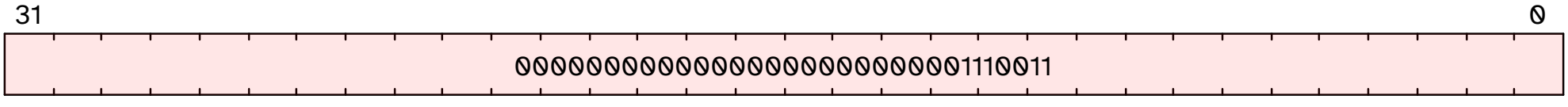
## C.45. ecall

**Environment call**

This instruction is defined by:

- I, version >= I@2.1.0

### C.45.1. Encoding




### C.45.2. Description

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.

*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.

### C.45.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.45.4. Decode Variables

### C.45.5. IDL Operation

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

### C.45.6. Sail Operation

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

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

### C.45.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

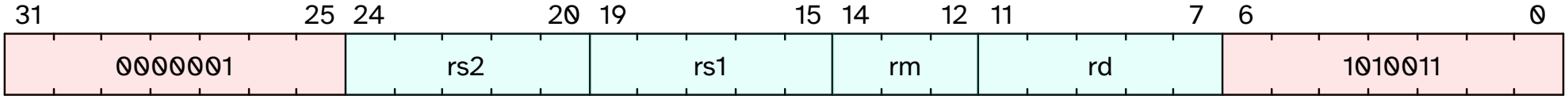
## C.46. fadd.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.46.1. Encoding




### C.46.2. Description

No description available.

### C.46.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.46.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.46.5. IDL Operation

```

```

### C.46.6. Exceptions

This instruction does not generate synchronous exceptions.

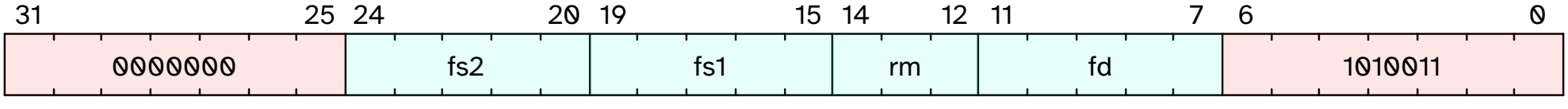
## C.47. fadd.s

**Single-precision floating-point addition**

This instruction is defined by:

- F, version >= F@2.2.0

### C.47.1. Encoding




### C.47.2. Description

Do the single-precision floating-point addition of fs1 and fs2 and store the result in fd. rm is the dynamic Rounding Mode.

### C.47.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.47.4. Decode Variables

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

### C.47.5. IDL Operation

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

### C.47.6. Sail Operation

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

### C.47.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.48. fclass.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.48.1. Encoding


### C.48.2. Description

No description available.

### C.48.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.48.4. Decode Variables

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

### C.48.5. IDL Operation

```

```

### C.48.6. Exceptions

This instruction does not generate synchronous exceptions.

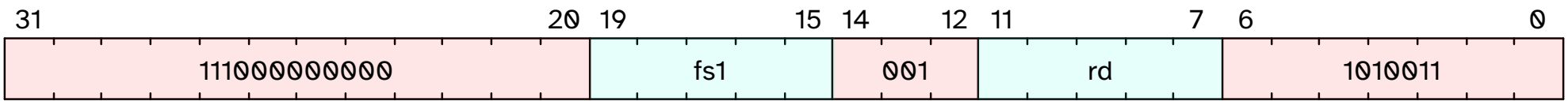
## C.49. fclass.s

### Single-precision floating-point classify

This instruction is defined by:

- F, version >= F@2.2.0

### C.49.1. Encoding




### C.49.2. Description

The fclass.s instruction examines the value in floating-point register *fs1* and writes to integer register *rd* a 10-bit mask that indicates the class of the floating-point number. The format of the mask is described in the table below. The corresponding bit in *rd* will be set if the property is true and clear otherwise. All other bits in *rd* are cleared. Note that exactly one bit in rd will be set. fclass.s does not set the floating-point exception flags.

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

| *rd* bit | Meaning |
|---|---|
| 0 | *rs1* is `-\infty`. |
| 1 | *rs1* is a negative normal number. |
| 2 | *rs1* is a negative subnormal number. |
| 3 | *rs1* is `-0`. |
| 4 | *rs1* is `+0`. |
| 5 | *rs1* is a positive subnormal number. |
| 6 | *rs1* is a positive normal number. |
| 7 | *rs1* is `+\infty`. |
| 8 | *rs1* is a signaling NaN. |
| 9 | *rs1* is a quiet NaN. |

### C.49.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.49.4. Decode Variables

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

### C.49.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = f[fs1][31:0];
if (is_sp_neg_inf?(sp_value)) {
  X[rd] = 1 << 0;
} else if (is_sp_neg_norm?(sp_value)) {
  X[rd] = 1 `<< 1;
} else if (is_sp_neg_subnorm?(sp_value)) {
  X[rd] = 1 `<< 2;
} else if (is_sp_neg_zero?(sp_value)) {
  X[rd] = 1 `<< 3;
} else if (is_sp_pos_zero?(sp_value)) {
  X[rd] = 1 `<< 4;
} else if (is_sp_pos_subnorm?(sp_value)) {
  X[rd] = 1 `<< 5;
} else if (is_sp_pos_norm?(sp_value)) {
  X[rd] = 1 `<< 6;
} else if (is_sp_pos_inf?(sp_value)) {
  X[rd] = 1 `<< 7;
} else if (is_sp_signaling_nan?(sp_value)) {
  X[rd] = 1 `<< 8;
```

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

### C.49.6. Sail Operation

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

### C.49.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

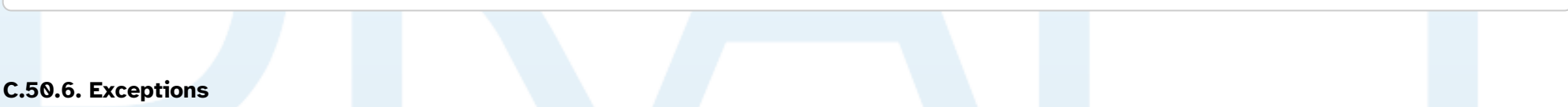
## C.50. fcvt.d.h

**No synopsis available**

This instruction is defined by:

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

### C.50.1. Encoding




### C.50.2. Description

No description available.

### C.50.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.50.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.50.5. IDL Operation

```

```

### C.50.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.51. fcvt.d.l

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.51.1. Encoding


### C.51.2. Description

No description available.

### C.51.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.51.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.51.5. IDL Operation

```

```

### C.51.6. Exceptions

This instruction does not generate synchronous exceptions.

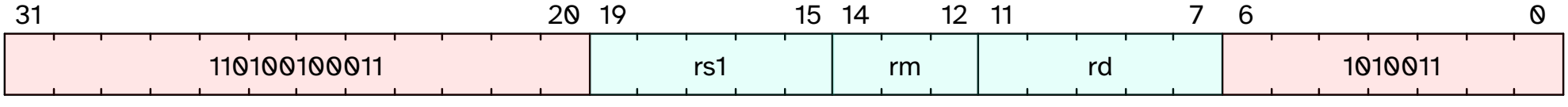
## C.52. fcvt.d.lu

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.52.1. Encoding




### C.52.2. Description

No description available.

### C.52.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.52.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.52.5. IDL Operation

```

```

### C.52.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.53. fcvt.d.s

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.53.1. Encoding


### C.53.2. Description

No description available.

### C.53.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.53.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.53.5. IDL Operation

```

```

### C.53.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.54. fcvt.d.w

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.54.1. Encoding


### C.54.2. Description

No description available.

### C.54.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.54.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.54.5. IDL Operation

```

```

### C.54.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.55. fcvt.d.wu

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.55.1. Encoding


### C.55.2. Description

No description available.

### C.55.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.55.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.55.5. IDL Operation

```

```

### C.55.6. Exceptions

This instruction does not generate synchronous exceptions.

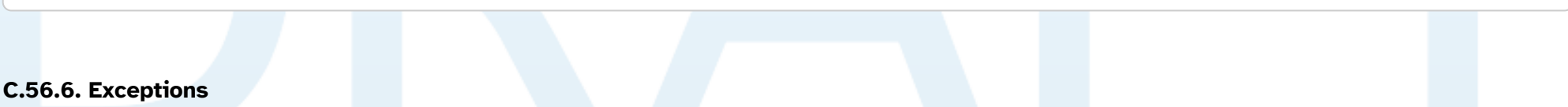
## C.56. fcvt.h.d

**No synopsis available**

This instruction is defined by:

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

### C.56.1. Encoding




### C.56.2. Description

No description available.

### C.56.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.56.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.56.5. IDL Operation

```

```

### C.56.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.57. fcvt.l.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.57.1. Encoding


### C.57.2. Description

No description available.

### C.57.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.57.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.57.5. IDL Operation

```

```

### C.57.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.58. fcvt.l.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.58.1. Encoding


### C.58.2. Description

No description available.

### C.58.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.58.4. Decode Variables

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

### C.58.5. IDL Operation

```

```

### C.58.6. Sail Operation

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

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

### C.58.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.59. fcvt.lu.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.59.1. Encoding


### C.59.2. Description

No description available.

### C.59.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.59.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.59.5. IDL Operation

```

```

### C.59.6. Exceptions

This instruction does not generate synchronous exceptions.

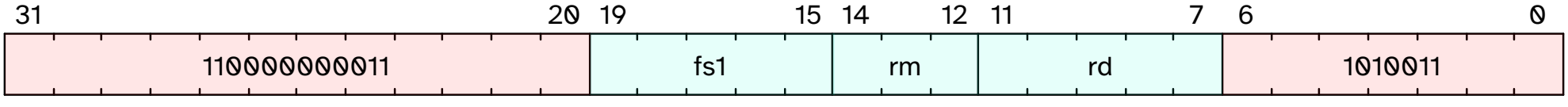
## C.60. fcvt.lu.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.60.1. Encoding




### C.60.2. Description

No description available.

### C.60.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.60.4. Decode Variables

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

### C.60.5. IDL Operation

```

```

### C.60.6. Sail Operation

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

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

### C.60.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.61. fcvt.s.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.61.1. Encoding


### C.61.2. Description

No description available.

### C.61.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.61.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.61.5. IDL Operation

```

```

### C.61.6. Exceptions

This instruction does not generate synchronous exceptions.

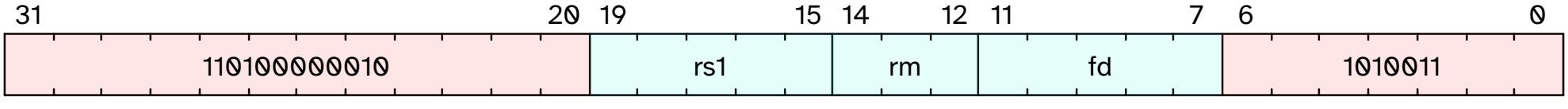
## C.62. fcvt.s.l

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.62.1. Encoding




### C.62.2. Description

No description available.

### C.62.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.62.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.62.5. IDL Operation

```

```

### C.62.6. Sail Operation

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

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

### C.62.7. Exceptions

This instruction does not generate synchronous exceptions.

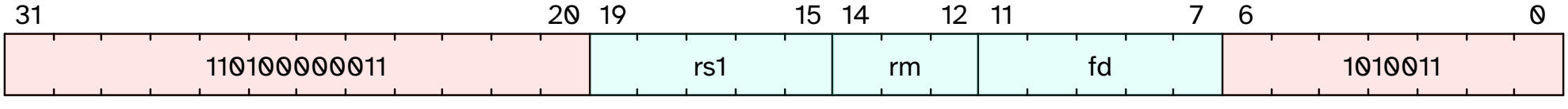
## C.63. fcvt.s.lu

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.63.1. Encoding




### C.63.2. Description

No description available.

### C.63.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.63.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.63.5. IDL Operation

```

```

### C.63.6. Sail Operation

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

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

### C.63.7. Exceptions

This instruction does not generate synchronous exceptions.

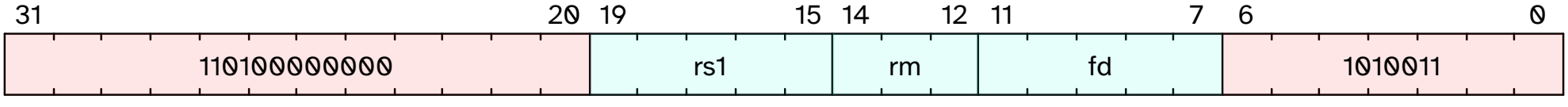
## C.64. fcvt.s.w

**Convert signed 32-bit integer to single-precision float**

This instruction is defined by:

- F, version >= F@2.2.0

### C.64.1. Encoding




### C.64.2. Description

Converts a 32-bit signed integer in integer register *rs1* into a floating-point number in floating-point register *fd*.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

### C.64.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.64.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.64.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
X[fd] = i32_to_f32(X[rs1], rounding_mode);
mark_f_state_dirty();
```

### C.64.6. Sail Operation

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

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

### C.64.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

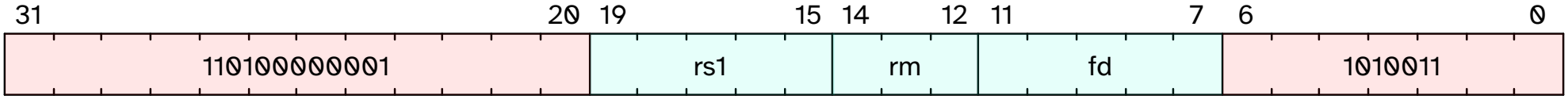
## C.65. fcvt.s.wu

**Convert unsigned 32-bit integer to single-precision float**

This instruction is defined by:

- F, version >= F@2.2.0

### C.65.1. Encoding




### C.65.2. Description

Converts a 32-bit unsigned integer in integer register *rs1* into a floating-point number in floating-point register *fd*.

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

### C.65.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.65.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.65.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
X[fd] = ui32_to_f32(X[rs1], rounding_mode);
mark_f_state_dirty();
```

### C.65.6. Sail Operation

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

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

### C.65.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.66. fcvt.w.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.66.1. Encoding


### C.66.2. Description

No description available.

### C.66.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.66.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.66.5. IDL Operation

```

```

### C.66.6. Exceptions

This instruction does not generate synchronous exceptions.

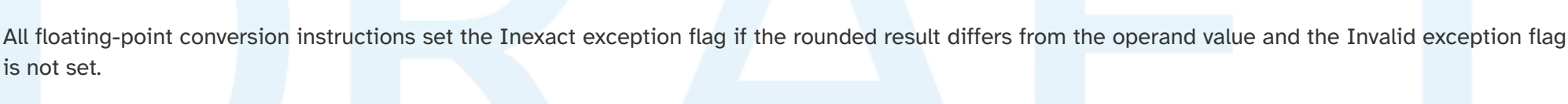
## C.67. fcvt.w.s

**Convert single-precision float to integer word to signed 32-bit integer**

This instruction is defined by:

- F, version >= F@2.2.0

### C.67.1. Encoding




### C.67.2. Description

Converts a floating-point number in floating-point register *fs1* to a signed 32-bit integer indicates integer register *rd*.

For XLEN >32, fcvt.w.s sign-extends the 32-bit result to the destination register width.

If the rounded result is not representable as a 32-bit signed integer, it is clipped to the nearest value and the invalid flag is set.

The range of valid inputs and behavior for invalid inputs are:

| | Value |
|---|---|
| **Minimum valid input (after rounding)** | `-2^31` |
| **Maximum valid input (after rounding)** | `2^31 - 1` |
| **Output for out-of-range negative input** | `-2^31` |
| **Output for <code>-∞</code>** | `-2^31` |
| **Output for out-of-range positive input** | `2^31 - 1` |
| **Output for <code>+∞</code> for <code>NaN</code>** | `2^31 - 1` |

All floating-point to integer and integer to floating-point conversion instructions round according to the *rm* field. A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

All floating-point conversion instructions set the Inexact exception flag if the rounded result differs from the operand value and the Invalid exception flag is not set.

### C.67.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.67.4. Decode Variables

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

### C.67.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = f[fs1][31:0];
Bits<1> sign = sp_value[31];
Bits<8> exp = sp_value[30:23];
Bits<23> sig = sp_value[22:0];
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
if ((exp == 0xff) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  X[rd] = SP_CANONICAL_NAN;
} else {
  if (exp != 0) {
    sig = sig | 0x00800000;
  }
  Bits<64> sig64 = sig << 32;
  Bits<16> shift_dist = 0xAA - exp;
  if (0 < shift_dist) {
```

```
    sig64 = softfloat_shiftRightJam64(sig64, shift_dist);
  }
  X[rd] = softfloat_roundToI32(sign, sig64, rounding_mode);
}
```

### C.67.6. Sail Operation

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

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

### C.67.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.68. fcvt.wu.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.68.1. Encoding


### C.68.2. Description

No description available.

### C.68.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.68.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.68.5. IDL Operation

```

```

### C.68.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.69. fcvt.wu.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.69.1. Encoding


### C.69.2. Description

No description available.

### C.69.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.69.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.69.5. IDL Operation

```

```

### C.69.6. Sail Operation

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

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

### C.69.7. Exceptions

This instruction does not generate synchronous exceptions.

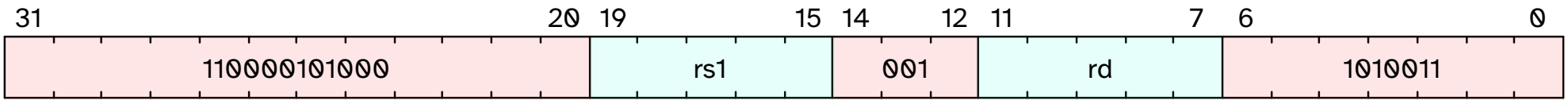
## C.70. fcvtmod.w.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.70.1. Encoding




### C.70.2. Description

No description available.

### C.70.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.70.4. Decode Variables

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

### C.70.5. IDL Operation

```

```

### C.70.6. Exceptions

This instruction does not generate synchronous exceptions.

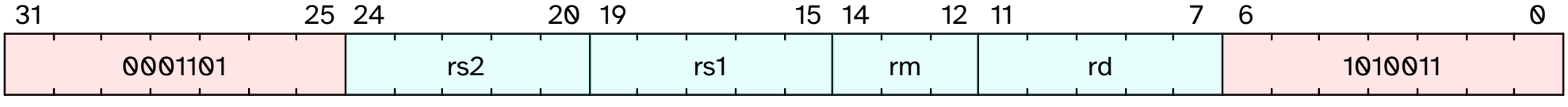
## C.71. fdiv.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.71.1. Encoding




### C.71.2. Description

No description available.

### C.71.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.71.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.71.5. IDL Operation

```

```

### C.71.6. Exceptions

This instruction does not generate synchronous exceptions.

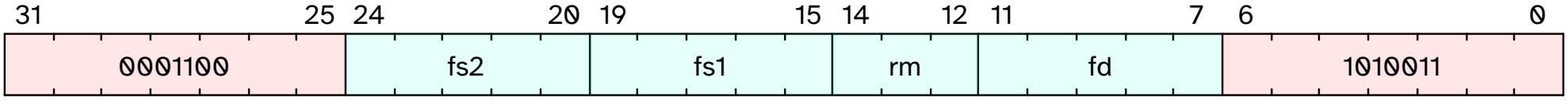
## C.72. fdiv.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.72.1. Encoding




### C.72.2. Description

No description available.

### C.72.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.72.4. Decode Variables

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

### C.72.5. IDL Operation

```

```

### C.72.6. Sail Operation

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

### C.72.7. Exceptions

This instruction does not generate synchronous exceptions.

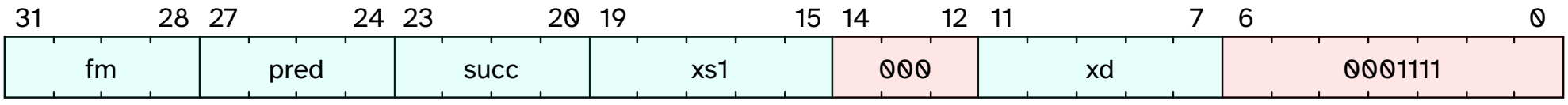
## C.73. fence

### Memory oxdering fence

This instruction is defined by:

- I, version >= I@2.1.0

### C.73.1. Encoding




### C.73.2. Description

Oxders memory operations.

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

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

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

*Table 10. Fence mode encoding*

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

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

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

The `xs1` and `xd` fields are unused and ignored.

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

*Table 11. Effective PR/PW/SR/SW in (H)S-mode*

*Table 12. Effective PR/PW/SR/SW in U-mode*

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

<%- end -%>

### C.73.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.73.4. Decode Variables

```
Bits<4> fm = $encoding[31:28];
Bits<4> pred = $encoding[27:24];
Bits<4> succ = $encoding[23:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

### C.73.5. IDL Operation

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

### C.73.6. Sail Operation

```
{
  // If the FIOM bit in menvcfg/senvcfg is set then the I/O bits can imply R/W.
```

```
  let fiom = is_fiom_active();
  let pred = effective_fence_set(pred, fiom);
  let succ = effective_fence_set(succ, fiom);

  match (pred, succ) {
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b11) => __barrier(Barrier_RISCV_rw_rw()),
    (_ : bits(2) @ 0b10, _ : bits(2) @ 0b11) => __barrier(Barrier_RISCV_r_rw()),
    (_ : bits(2) @ 0b10, _ : bits(2) @ 0b10) => __barrier(Barrier_RISCV_r_r()),
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b01) => __barrier(Barrier_RISCV_rw_w()),
    (_ : bits(2) @ 0b01, _ : bits(2) @ 0b01) => __barrier(Barrier_RISCV_w_w()),
    (_ : bits(2) @ 0b01, _ : bits(2) @ 0b11) => __barrier(Barrier_RISCV_w_rw()),
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b10) => __barrier(Barrier_RISCV_rw_r()),
    (_ : bits(2) @ 0b10, _ : bits(2) @ 0b01) => __barrier(Barrier_RISCV_r_w()),
    (_ : bits(2) @ 0b01, _ : bits(2) @ 0b10) => __barrier(Barrier_RISCV_w_r()),

    (_ : bits(4)       , _ : bits(2) @ 0b00) => (),
    (_ : bits(2) @ 0b00, _ : bits(4)       ) => (),

    _ => { print("FIXME: unsupported fence");
           () }
  };
  RETIRE_SUCCESS
}
```

### C.73.7. Exceptions

This instruction does not generate synchronous exceptions.

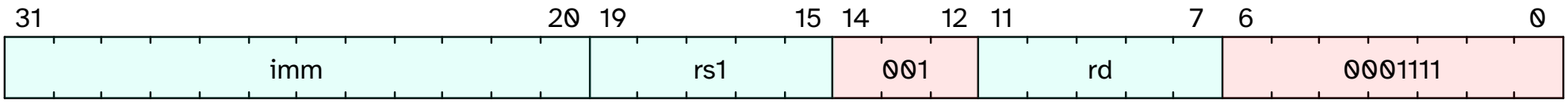
## C.74. fence.i

### Instruction fence

This instruction is defined by:

- Zifencei, version >= Zifencei@2.0.0

### C.74.1. Encoding




### C.74.2. Description

The FENCE.I instruction is used to synchronize the instruction and data streams. RISC-V does not guarantee that stores to instruction memory will be made visible to instruction fetches on a RISC-V hart until that hart executes a FENCE.I instruction. A FENCE.I instruction ensures that a subsequent instruction fetch on a RISC-V hart will see any previous data stores already visible to the same RISC-V hart. FENCE.I does *not* ensure that other RISC-V harts' instruction fetches will observe the local hart's stores in a multiprocessor system. To make a store to instruction memory visible to all RISC-V harts, the writing hart also has to execute a data FENCE before requesting that all remote RISC-V harts execute a FENCE.I.

The unused fields in the FENCE.I instruction, *imm[11:0]*, *rs1*, and *rd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

*Because FENCE.I only orders stores with a hart's own instruction fetches, application code should only rely upon FENCE.I if the application thread will not be migrated to a different hart. The EEI can provide mechanisms for efficient multiprocessor instruction-stream synchronization.*

### C.74.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.74.4. Decode Variables

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

### C.74.5. IDL Operation

```
ifence();
```

### C.74.6. Sail Operation

```
{ /* __barrier(Barrier_RISCV_i); */ RETIRE_SUCCESS }
```

### C.74.7. Exceptions

This instruction does not generate synchronous exceptions.

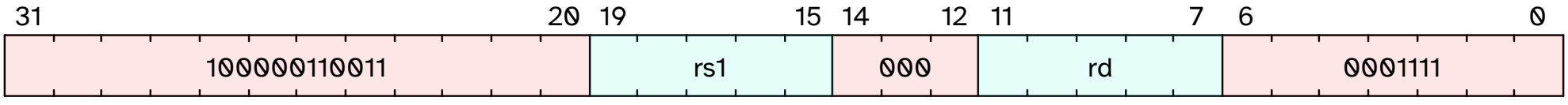
## C.75. fence.tso

**Memory ordering fence, total store ordering**

This instruction is defined by:

- I, version >= I@2.1.0

### C.75.1. Encoding




### C.75.2. Description

Orders memory operations.

fence.tso orders all load operations in its predecessor set before all memory operations in its successor set, and all store operations in its predecessor set before all store operations in its successor set. This leaves non-AMO store operations in the 'fence.tso's predecessor set unordered with non-AMO loads in its successor set.

The `rs1` and `rd` fields are unused and ignored.

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

### C.75.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.75.4. Decode Variables

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

### C.75.5. IDL Operation

```
fence_tso();
```

### C.75.6. Sail Operation

```
{
  match (pred, succ) {
    (_ : bits(2) @ 0b11, _ : bits(2) @ 0b11) => sail_barrier(Barrier_RISCV_tso),
    (_ : bits(2) @ 0b00, _ : bits(2) @ 0b00) => (),

    _ => { print("FIXME: unsupported fence");
           () }
  };
  RETIRE_SUCCESS
}
```

### C.75.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.76. feq.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.76.1. Encoding


### C.76.2. Description

No description available.

### C.76.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.76.4. Decode Variables

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

### C.76.5. IDL Operation

```

```

### C.76.6. Exceptions

This instruction does not generate synchronous exceptions.

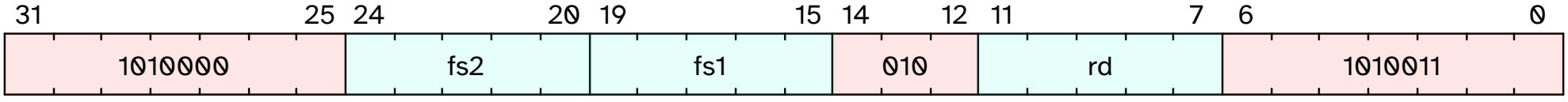
## C.77. feq.s

**Single-precision floating-point equal**

This instruction is defined by:

- F, version >= F@2.2.0

### C.77.1. Encoding




### C.77.2. Description

Writes 1 to *rd* if *fs1* and *fs2* are equal, and 0 otherwise.

If either operand is NaN, the result is 0 (not equal). If either operand is a signaling NaN, the invalid flag is set.

Positive zero is considered equal to negative zero.

### C.77.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.77.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.77.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs1][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  if (is_sp_signaling_nan?(sp_value_a) || is_sp_signaling_nan?(sp_value_b)) {
    set_fp_flag(FpFlag::NV);
  }
  X[rd] = 0;
} else {
  X[rd] = sp_value_a == sp_value_b) || ((sp_value_a | sp_value_b)[30:0] == 0 ? 1 : 0;
}
```

### C.77.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.77.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.78. fld

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.78.1. Encoding


### C.78.2. Description

No description available.

### C.78.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.78.4. Decode Variables

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

### C.78.5. IDL Operation

```

```

### C.78.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.79. fle.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.79.1. Encoding


### C.79.2. Description

No description available.

### C.79.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.79.4. Decode Variables

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

### C.79.5. IDL Operation

```

```

### C.79.6. Exceptions

This instruction does not generate synchronous exceptions.

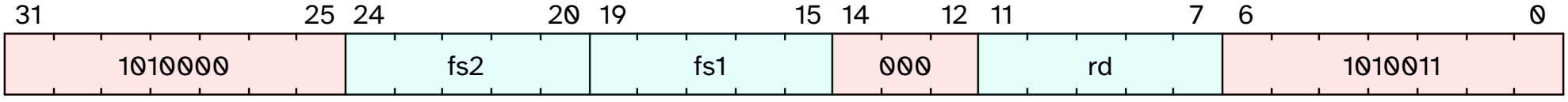
## C.80. fle.s

### Single-precision floating-point less than or equal

This instruction is defined by:

- F, version >= F@2.2.0

### C.80.1. Encoding




### C.80.2. Description

Writes 1 to *rd* if *fs1* is less than or equal to *fs2*, and 0 otherwise.

If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set.

Positive zero and negative zero are considered equal.

### C.80.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.80.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.80.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs2][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  if (is_sp_signaling_nan?(sp_value_a) || is_sp_signaling_nan?(sp_value_b)) {
    set_fp_flag(FpFlag::NV);
  }
  X[rd] = 0;
} else {
  X[rd] = sp_value_a == sp_value_b) || ((sp_value_a | sp_value_b)[30:0] == 0 ? 1 : 0;
}
```

### C.80.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.80.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

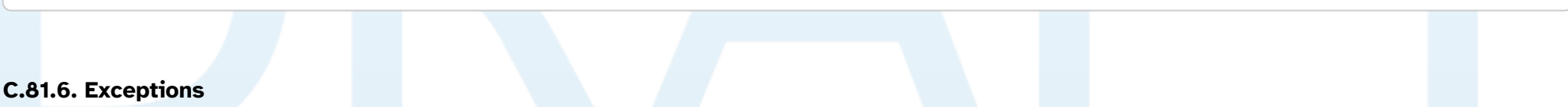
## C.81. fleq.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.81.1. Encoding




### C.81.2. Description

No description available.

### C.81.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.81.4. Decode Variables

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

### C.81.5. IDL Operation

```

```

### C.81.6. Exceptions

This instruction does not generate synchronous exceptions.

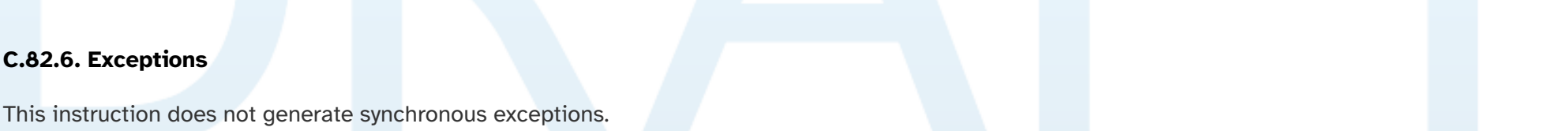
## C.82. fli.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.82.1. Encoding




### C.82.2. Description

No description available.

### C.82.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.82.4. Decode Variables

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

### C.82.5. IDL Operation

```

```

### C.82.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.83. flt.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.83.1. Encoding


### C.83.2. Description

No description available.

### C.83.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.83.4. Decode Variables

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

### C.83.5. IDL Operation

```

```

### C.83.6. Exceptions

This instruction does not generate synchronous exceptions.

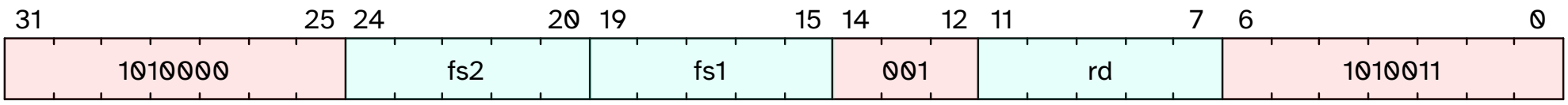
## C.84. flt.s

**Single-precision floating-point less than**

This instruction is defined by:

- F, version >= F@2.2.0

### C.84.1. Encoding




### C.84.2. Description

Writes 1 to *rd* if *fs1* is less than *fs2*, and 0 otherwise.

If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set.

### C.84.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.84.4. Decode Variables

```
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.84.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value_a = f[fs1][31:0];
Bits<32> sp_value_b = f[fs2][31:0];
if (is_sp_nan?(sp_value_a) || is_sp_nan?(sp_value_b)) {
  set_fp_flag(FpFlag::NV);
  X[rd] = 0;
} else {
  Boolean sign_a = sp_value_a[31] == 1;
  Boolean sign_b = sp_value_b[31] == 1;
  Boolean a_lt_b = (sign_a != sign_b) ? (sign_a && sp_value_a[30:0] | sp_value_b[30:0]) != 0 : sp_value_a !=
sp_value_b) && (sign_a != (sp_value_a < sp_value_b);
  X[rd] = a_lt_b ? 1 : 0;
}
```

### C.84.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.84.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

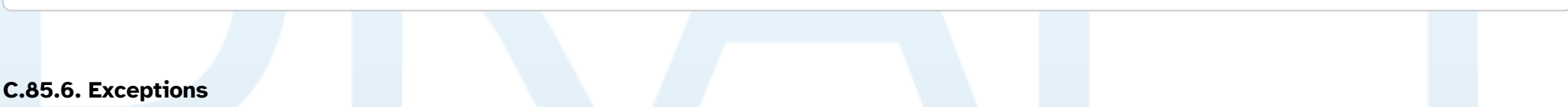
## C.85. fltq.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.85.1. Encoding




### C.85.2. Description

No description available.

### C.85.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.85.4. Decode Variables

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

### C.85.5. IDL Operation

```

```

### C.85.6. Exceptions

This instruction does not generate synchronous exceptions.

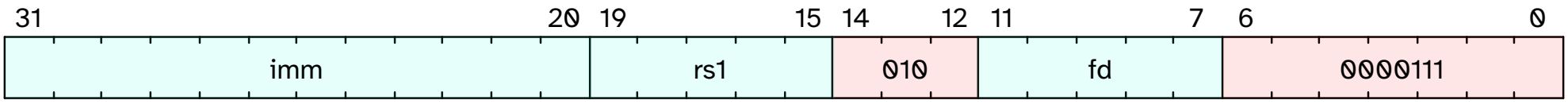
## C.86. flw

**Single-precision floating-point load**

This instruction is defined by:

- F, version >= F@2.2.0

### C.86.1. Encoding




### C.86.2. Description

The flw instruction loads a single-precision floating-point value from memory at address *rs1* + *imm* into floating-point register *fd*.

flw does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

### C.86.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.86.4. Decode Variables

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

### C.86.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $signed(imm);
Bits<32> sp_value = read_memory<32>(virtual_address, $encoding);
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

### C.86.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(rs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(rs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(addr, _) => {
          let (aq, rl, res) = (false, false, false);
          match (width) {
            BYTE => { handle_illegal(); RETIRE_FAIL },
            HALF =>
               process_fload16(rd, vaddr, mem_read(Read(Data), addr, 2, aq, rl, res)),
            WORD =>
               process_fload32(rd, vaddr, mem_read(Read(Data), addr, 4, aq, rl, res)),
            DOUBLE if sizeof(flen) >= 64 =>
               process_fload64(rd, vaddr, mem_read(Read(Data), addr, 8, aq, rl, res)),
            _ => report_invalid_width(__FILE__, __LINE__, width, "floating point load"),
          }
```

```
        }
      }
    }
  }
```

### C.86.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

## C.87. fmadd.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.87.1. Encoding




### C.87.2. Description

No description available.

### C.87.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.87.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.87.5. IDL Operation

```

```

### C.87.6. Exceptions

This instruction does not generate synchronous exceptions.

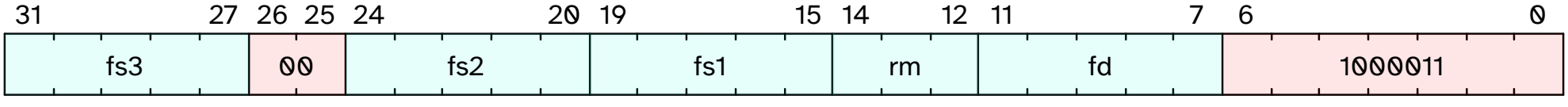
## C.88. fmadd.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.88.1. Encoding




### C.88.2. Description

No description available.

### C.88.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.88.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.88.5. IDL Operation

```

```

### C.88.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

### C.88.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.89. fmax.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.89.1. Encoding


### C.89.2. Description

No description available.

### C.89.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.89.4. Decode Variables

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

### C.89.5. IDL Operation

```

```

### C.89.6. Exceptions

This instruction does not generate synchronous exceptions.

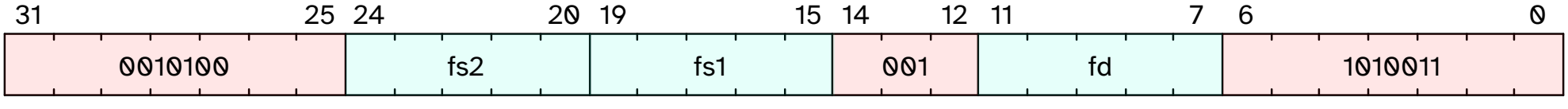
## C.90. fmax.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.90.1. Encoding




### C.90.2. Description

No description available.

### C.90.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.90.4. Decode Variables

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

### C.90.5. IDL Operation

```

```

### C.90.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.90.7. Exceptions

This instruction does not generate synchronous exceptions.

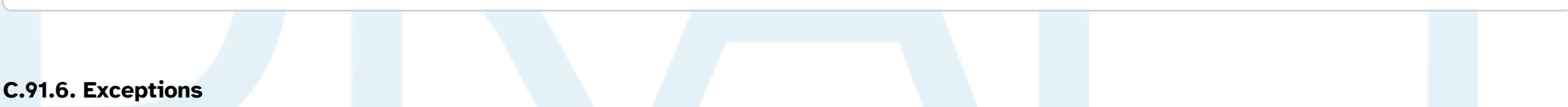
## C.91. fmaxm.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.91.1. Encoding




### C.91.2. Description

No description available.

### C.91.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.91.4. Decode Variables

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

### C.91.5. IDL Operation

```

```

### C.91.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.92. fmin.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.92.1. Encoding


### C.92.2. Description

No description available.

### C.92.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.92.4. Decode Variables

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

### C.92.5. IDL Operation

```

```

### C.92.6. Exceptions

This instruction does not generate synchronous exceptions.

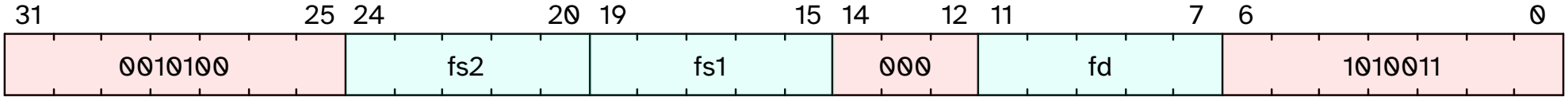
## C.93. fmin.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.93.1. Encoding




### C.93.2. Description

No description available.

### C.93.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.93.4. Decode Variables

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

### C.93.5. IDL Operation

```

```

### C.93.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.93.7. Exceptions

This instruction does not generate synchronous exceptions.

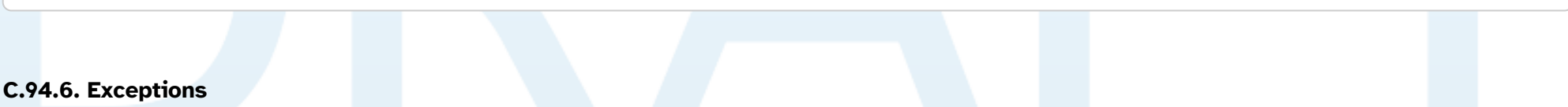
## C.94. fminm.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.94.1. Encoding




### C.94.2. Description

No description available.

### C.94.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.94.4. Decode Variables

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

### C.94.5. IDL Operation

```

```

### C.94.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.95. fmsub.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.95.1. Encoding




### C.95.2. Description

No description available.

### C.95.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.95.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.95.5. IDL Operation

```

```

### C.95.6. Exceptions

This instruction does not generate synchronous exceptions.

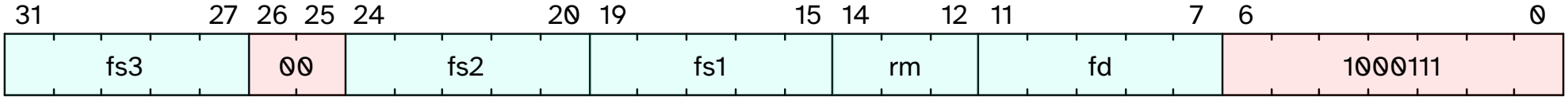
## C.96. fmsub.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.96.1. Encoding




### C.96.2. Description

No description available.

### C.96.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.96.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.96.5. IDL Operation

```

```

### C.96.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

### C.96.7. Exceptions

This instruction does not generate synchronous exceptions.

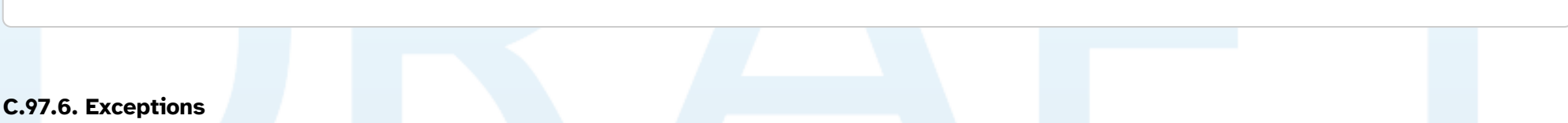
## C.97. fmul.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.97.1. Encoding




### C.97.2. Description

No description available.

### C.97.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.97.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.97.5. IDL Operation

```

```

### C.97.6. Exceptions

This instruction does not generate synchronous exceptions.

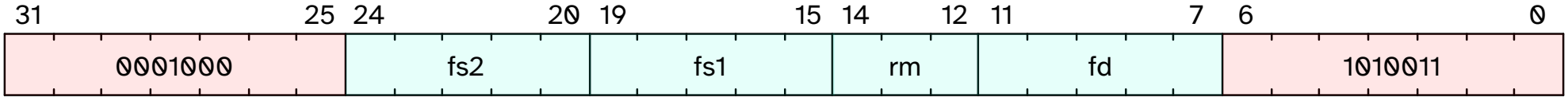
## C.98. fmul.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.98.1. Encoding




### C.98.2. Description

No description available.

### C.98.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.98.4. Decode Variables

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

### C.98.5. IDL Operation

```

```

### C.98.6. Sail Operation

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

### C.98.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.99. fmv.d.x

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.99.1. Encoding


### C.99.2. Description

No description available.

### C.99.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.99.4. Decode Variables

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

### C.99.5. IDL Operation

```

```

### C.99.6. Exceptions

This instruction does not generate synchronous exceptions.

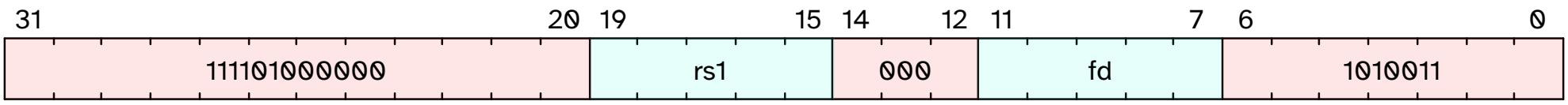
## C.100. fmv.h.x

### Half-precision floating-point move from integer

This instruction is defined by:

- F, version >= F@2.2.0

### C.100.1. Encoding




### C.100.2. Description

Moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register `rs1` to the floating-point register `fd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

### C.100.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.100.4. Decode Variables

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

### C.100.5. IDL Operation

```
check_f_ok($encoding);
Bits<16> hp_value = X[rs1][15:0];
f[fd] = nan_box<16, FLEN>(hp_value);
mark_f_state_dirty();
```

### C.100.6. Sail Operation

```
{
  let rs1_val_X            = X(rs1);
  let rd_val_H             = rs1_val_X [15..0];
  F(rd) = nan_box (rd_val_H);
  RETIRE_SUCCESS
}
```

### C.100.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

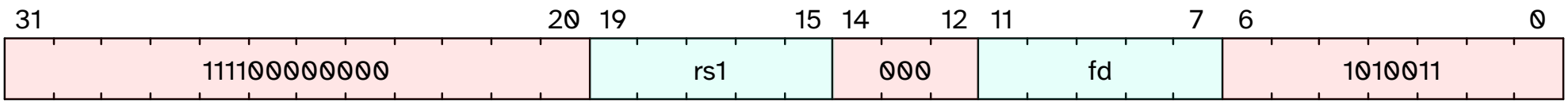
## C.101. fmv.w.x

**Single-precision floating-point move from integer**

This instruction is defined by:

- F, version >= F@2.2.0

### C.101.1. Encoding




### C.101.2. Description

Moves the single-precision value encoded in IEEE 754-2008 standard encoding from the lower 32 bits of integer register `rs1` to the floating-point register `fd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

### C.101.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.101.4. Decode Variables

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

### C.101.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = X[rs1][31:0];
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

### C.101.6. Sail Operation

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

### C.101.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.102. fmv.x.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.102.1. Encoding


### C.102.2. Description

No description available.

### C.102.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.102.4. Decode Variables

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

### C.102.5. IDL Operation

```

```

### C.102.6. Exceptions

This instruction does not generate synchronous exceptions.

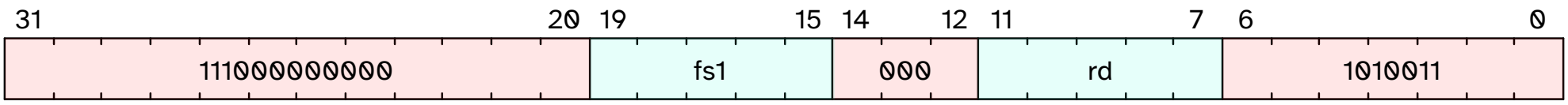
## C.103. fmv.x.w

**Move single-precision value from floating-point to integer register**

This instruction is defined by:

- F, version >= F@2.2.0

### C.103.1. Encoding




### C.103.2. Description

Moves the single-precision value in floating-point register rs1 represented in IEEE 754-2008 encoding to the lower 32 bits of integer register rd. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.

### C.103.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.103.4. Decode Variables

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

### C.103.5. IDL Operation

```
check_f_ok($encoding);
X[rd] = sext(f[fs1][31:0], 32);
```

### C.103.6. Sail Operation

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

### C.103.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

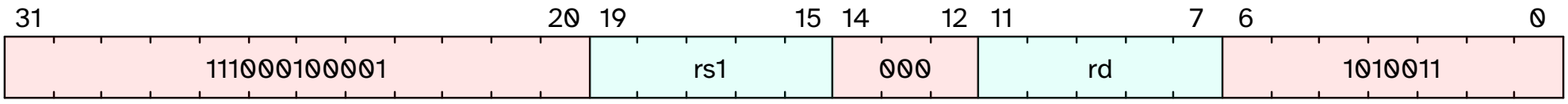
## C.104. fmvh.x.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.104.1. Encoding




### C.104.2. Description

No description available.

### C.104.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.104.4. Decode Variables

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

### C.104.5. IDL Operation

```

```

### C.104.6. Exceptions

This instruction does not generate synchronous exceptions.

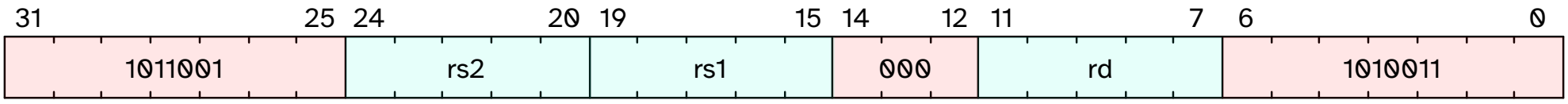
## C.105. fmvp.d.x

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.105.1. Encoding




### C.105.2. Description

No description available.

### C.105.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.105.4. Decode Variables

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

### C.105.5. IDL Operation

```

```

### C.105.6. Exceptions

This instruction does not generate synchronous exceptions.

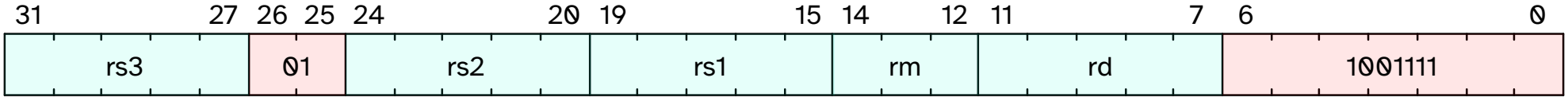
## C.106. fnmadd.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.106.1. Encoding




### C.106.2. Description

No description available.

### C.106.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.106.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.106.5. IDL Operation

```

```

### C.106.6. Exceptions

This instruction does not generate synchronous exceptions.

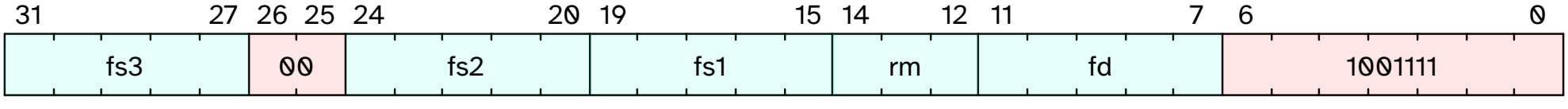
## C.107. fnmadd.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.107.1. Encoding




### C.107.2. Description

No description available.

### C.107.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.107.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.107.5. IDL Operation

```

```

### C.107.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

### C.107.7. Exceptions

This instruction does not generate synchronous exceptions.

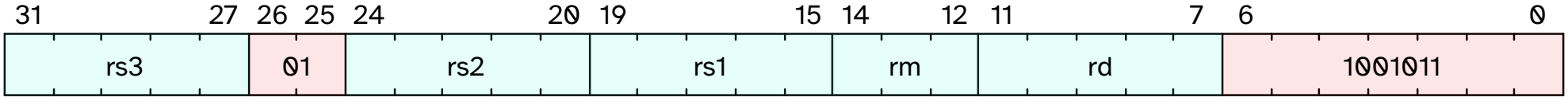
## C.108. fnmsub.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.108.1. Encoding




### C.108.2. Description

No description available.

### C.108.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.108.4. Decode Variables

```
Bits<5> rs3 = $encoding[31:27];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.108.5. IDL Operation

```

```

### C.108.6. Exceptions

This instruction does not generate synchronous exceptions.

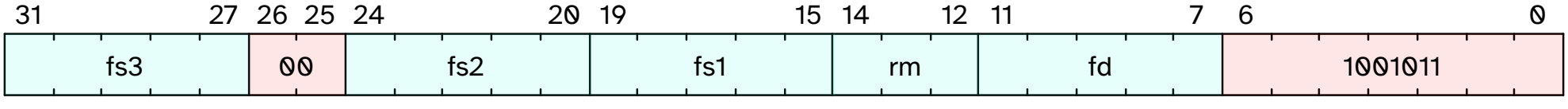
## C.109. fnmsub.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.109.1. Encoding




### C.109.2. Description

No description available.

### C.109.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.109.4. Decode Variables

```
Bits<5> fs3 = $encoding[31:27];
Bits<5> fs2 = $encoding[24:20];
Bits<5> fs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> fd = $encoding[11:7];
```

### C.109.5. IDL Operation

```

```

### C.109.6. Sail Operation

```
{
  let rs1_val_32b = F_or_X_S(rs1);
  let rs2_val_32b = F_or_X_S(rs2);
  let rs3_val_32b = F_or_X_S(rs3);
  match (select_instr_or_fcsr_rm (rm)) {
    None() => { handle_illegal(); RETIRE_FAIL },
    Some(rm') => {
      let rm_3b = encdec_rounding_mode(rm');
      let (fflags, rd_val_32b) : (bits(5), bits(32)) =
        match op {
          FMADD_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, rs3_val_32b),
          FMSUB_S  => riscv_f32MulAdd (rm_3b, rs1_val_32b, rs2_val_32b, negate_S (rs3_val_32b)),
          FNMSUB_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, rs3_val_32b),
          FNMADD_S => riscv_f32MulAdd (rm_3b, negate_S (rs1_val_32b), rs2_val_32b, negate_S (rs3_val_32b))
        };
      accrue_fflags(fflags);
      F_or_X_S(rd) = rd_val_32b;
      RETIRE_SUCCESS
    }
  }
}
```

### C.109.7. Exceptions

This instruction does not generate synchronous exceptions.

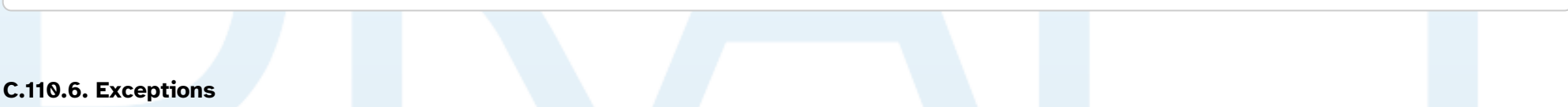
## C.110. fround.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.110.1. Encoding




### C.110.2. Description

No description available.

### C.110.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.110.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.110.5. IDL Operation

```

```

### C.110.6. Exceptions

This instruction does not generate synchronous exceptions.

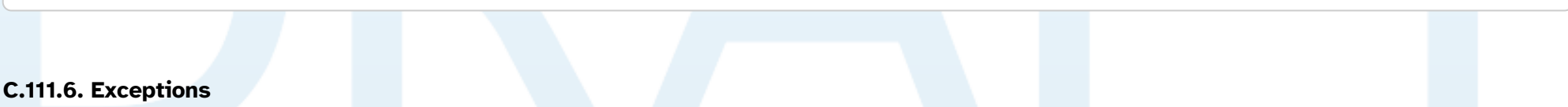
## C.111. froundnx.d

**No synopsis available**

This instruction is defined by:

- allOf:
  - D, version >= D@2.2.0
  - Zfa, version >= Zfa@1.0.0

### C.111.1. Encoding




### C.111.2. Description

No description available.

### C.111.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.111.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.111.5. IDL Operation

```

```

### C.111.6. Exceptions

This instruction does not generate synchronous exceptions.

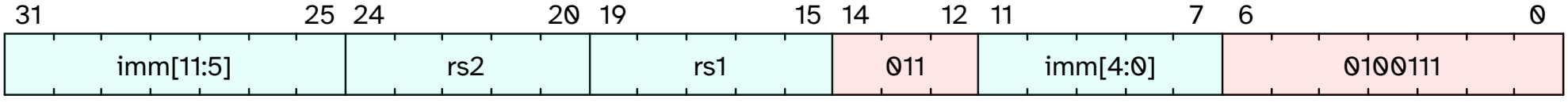
## C.112. fsd

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.112.1. Encoding




### C.112.2. Description

No description available.

### C.112.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.112.4. Decode Variables

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

### C.112.5. IDL Operation

```

```

### C.112.6. Exceptions

This instruction does not generate synchronous exceptions.

## C.113. fsgnj.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.113.1. Encoding


### C.113.2. Description

No description available.

### C.113.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.113.4. Decode Variables

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

### C.113.5. IDL Operation

```

```

### C.113.6. Exceptions

This instruction does not generate synchronous exceptions.

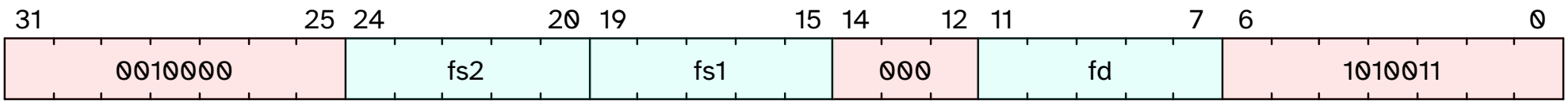
## C.114. fsgnj.s

### Single-precision sign inject

This instruction is defined by:

- F, version >= F@2.2.0

### C.114.1. Encoding




### C.114.2. Description

Writes *fd* with sign bit of *fs2* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

### C.114.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.114.4. Decode Variables

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

### C.114.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = {f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

### C.114.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.114.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.115. fsgnjn.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.115.1. Encoding


### C.115.2. Description

No description available.

### C.115.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.115.4. Decode Variables

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

### C.115.5. IDL Operation

```

```

### C.115.6. Exceptions

This instruction does not generate synchronous exceptions.

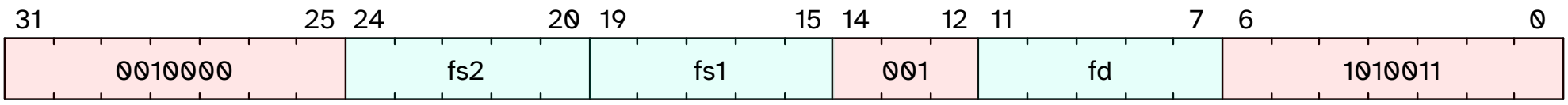
## C.116. fsgnjn.s

**Single-precision sign inject negate**

This instruction is defined by:

- F, version >= F@2.2.0

### C.116.1. Encoding




### C.116.2. Description

Writes *fd* with the opposite of the sign bit of *fs2* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

### C.116.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.116.4. Decode Variables

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

### C.116.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = {~f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

### C.116.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.116.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.117. fsgnjx.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.117.1. Encoding


### C.117.2. Description

No description available.

### C.117.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.117.4. Decode Variables

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

### C.117.5. IDL Operation

```

```

### C.117.6. Exceptions

This instruction does not generate synchronous exceptions.

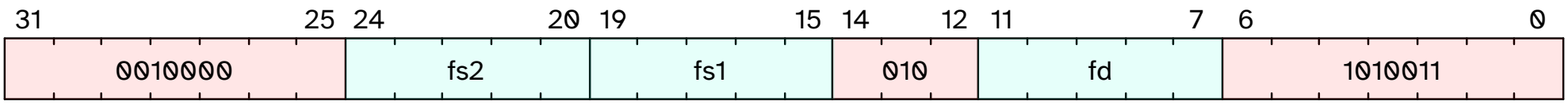
## C.118. fsgnjx.s

**Single-precision sign inject exclusive or**

This instruction is defined by:

- F, version >= F@2.2.0

### C.118.1. Encoding




### C.118.2. Description

Writes *fd* with the xor of the sign bits of *fs2* and *fs1* and the exponent and mantissa of *fs1*.

Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

### C.118.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.118.4. Decode Variables

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

### C.118.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = {f[fs1][31] ^ f[fs2][31], f[fs1][30:0]};
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

### C.118.6. Sail Operation

```
{
  let rs1_val_S = F_or_X_S(rs1);
  let rs2_val_S = F_or_X_S(rs2);

  let (fflags, rd_val) : (bits_fflags, bool) =
      riscv_f32Le (rs1_val_S, rs2_val_S);

  accrue_fflags(fflags);
  X(rd) = zero_extend(bool_to_bits(rd_val));
  RETIRE_SUCCESS
}
```

### C.118.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

## C.119. fsqrt.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.119.1. Encoding


### C.119.2. Description

No description available.

### C.119.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.119.4. Decode Variables

```
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.119.5. IDL Operation

```

```

### C.119.6. Exceptions

This instruction does not generate synchronous exceptions.

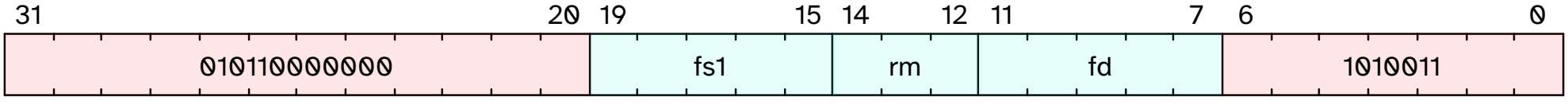
## C.120. fsqrt.s

**No synopsis available**

This instruction is defined by:

- F, version >= F@2.2.0

### C.120.1. Encoding




### C.120.2. Description

No description available.

### C.120.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.120.4. Decode Variables

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

### C.120.5. IDL Operation

```

```

### C.120.6. Sail Operation

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

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

### C.120.7. Exceptions

This instruction does not generate synchronous exceptions.

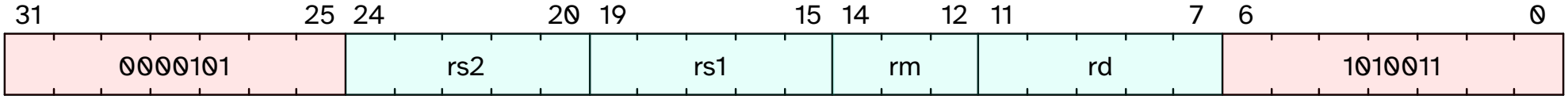
## C.121. fsub.d

**No synopsis available**

This instruction is defined by:

- D, version >= D@2.2.0

### C.121.1. Encoding




### C.121.2. Description

No description available.

### C.121.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.121.4. Decode Variables

```
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<3> rm = $encoding[14:12];
Bits<5> rd = $encoding[11:7];
```

### C.121.5. IDL Operation

```

```

### C.121.6. Exceptions

This instruction does not generate synchronous exceptions.

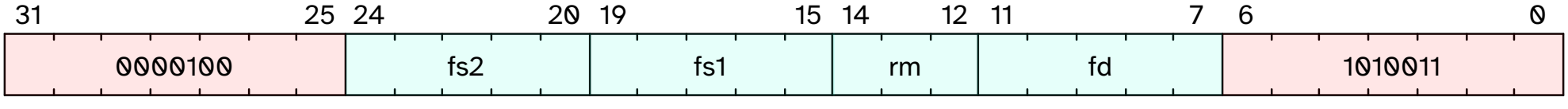
## C.122. fsub.s

**Single-precision floating-point subtraction**

This instruction is defined by:

- F, version >= F@2.2.0

### C.122.1. Encoding




### C.122.2. Description

Do the single-precision floating-point subtraction of fs2 from fs1 and store the result in fd. rm is the dynamic Rounding Mode.

### C.122.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.122.4. Decode Variables

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

### C.122.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode mode = rm_to_mode(rm, $encoding);
X[fd] = f32_sub(X[fs1], X[fs2], mode);
```

### C.122.6. Sail Operation

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

### C.122.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

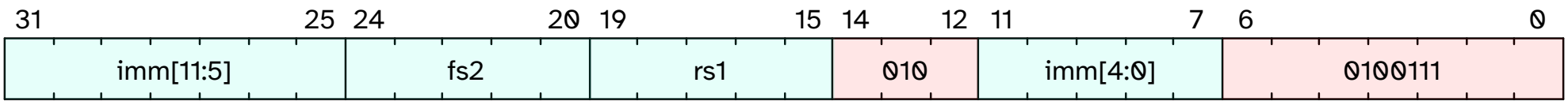
## C.123. fsw

**Single-precision floating-point store**

This instruction is defined by:

- F, version >= F@2.2.0

### C.123.1. Encoding




### C.123.2. Description

The fsw instruction stores a single-precision floating-point value in *fs2* to memory at address *rs1* + *imm*.

fsw does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

### C.123.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.123.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
```

### C.123.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[rs1] + $signed(imm);
write_memory<32>(virtual_address, f[fs2][31:0], $encoding);
```

### C.123.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  let (aq, rl, con) = (false, false, false);
  /* Get the address, X(rs1) + offset.
     Some extensions perform additional checks on address validity. */
match ext_data_get_addr(rs1, offset, Write(Data), width) {
  Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
  Ext_DataAddr_OK(vaddr) =>
    if   check_misaligned(vaddr, width)
    then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
    else match translateAddr(vaddr, Write(Data)) {
      TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
      TR_Address(addr, _) => {
        let eares : MemoryOpResult(unit) = match width {
          BYTE   => MemValue () /* bogus placeholder for illegal size */,
          HALF   => mem_write_ea(addr, 2, aq, rl, false),
          WORD   => mem_write_ea(addr, 4, aq, rl, false),
          DOUBLE => mem_write_ea(addr, 8, aq, rl, false)
        };
        match (eares) {
          MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
          MemValue(_) => {
            let rs2_val = F(rs2);
            match (width) {
              BYTE => { handle_illegal(); RETIRE_FAIL },
              HALF => process_fstore (vaddr, mem_write_value(addr, 2, rs2_val[15..0], aq, rl, con)),
              WORD => process_fstore (vaddr, mem_write_value(addr, 4, rs2_val[31..0], aq, rl, con)),
              DOUBLE if sizeof(flen) >= 64 =>
                process_fstore (vaddr, mem_write_value(addr, 8, rs2_val, aq, rl, con)),
```

```
                _ => report_invalid_width(__FILE__, __LINE__, width, "floating point store"),
              };
            }
          }
        }
      }
  }
}
```

### C.123.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

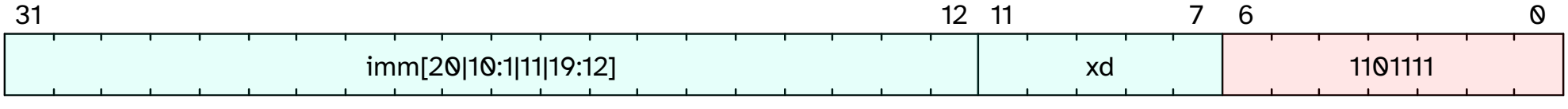
## C.124. jal

**Jump and link**

This instruction is defined by:

- I, version >= I@2.1.0

### C.124.1. Encoding




### C.124.2. Description

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

### C.124.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.124.4. Decode Variables

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

### C.124.5. IDL Operation

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

### C.124.6. Sail Operation

```
{
  let t : xlenbits = PC + sign_extend(imm);
  /* Extensions get the first checks on the prospective target address. */
  match ext_control_check_pc(t) {
    Ext_ControlAddr_Error(e) => {
      ext_handle_control_check_error(e);
      RETIRE_FAIL
    },
    Ext_ControlAddr_OK(target) => {
      /* Perform standaxd alignment check */
      if bit_to_bool(target[1]) & not(extension("C"))
      then {
        handle_mem_exception(target, E_Fetch_Addr_Align());
        RETIRE_FAIL
      } else {
        X(xd) = get_next_pc();
        set_next_pc(target);
        RETIRE_SUCCESS
      }
    }
  }
}
```

### C.124.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

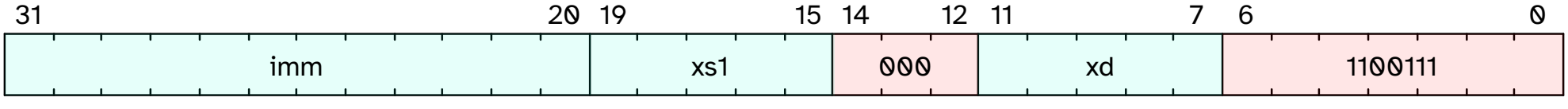
## C.125. jalr

**Jump and link register**

This instruction is defined by:

- I, version >= I@2.1.0

### C.125.1. Encoding




### C.125.2. Description

Jump to an address formed by adding xs1 to a signed offset then clearing the least significant bit, and store the return address in xd.

### C.125.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.125.4. Decode Variables

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

### C.125.5. IDL Operation

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

### C.125.6. Sail Operation

```
{
/* For the sequential model, the memory-model definition doesn't work directly
 * if xs1 = xd.  We would effectively have to keep a regfile for reads and another for
 * writes, and swap on instruction completion.  This could perhaps be optimized in
 * some manner, but for now, we just keep a reoxdered definition to improve simulator
 * performance.
 */
  let t : xlenbits = X(xs1) + sign_extend(imm);
  /* Extensions get the first checks on the prospective target address. */
  match ext_control_check_addr(t) {
    Ext_ControlAddr_Error(e) => {
      ext_handle_control_check_error(e);
      RETIRE_FAIL
    },
    Ext_ControlAddr_OK(addr) => {
      let target = [addr with 0 = bitzero];  /* clear addr[0] */
      if bit_to_bool(target[1]) & not(extension("C")) then {
        handle_mem_exception(target, E_Fetch_Addr_Align());
        RETIRE_FAIL
      } else {
        X(xd) = get_next_pc();
        set_next_pc(target);
        RETIRE_SUCCESS
      }
    }
  }
}
```

### C.125.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

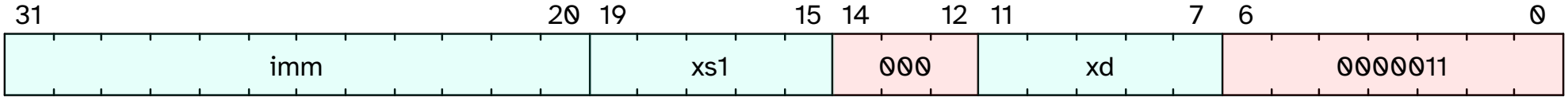
## C.126. lb

**Load byte**

This instruction is defined by:

- I, version >= I@2.1.0

### C.126.1. Encoding




### C.126.2. Description

Load 8 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Sign extend the result.

### C.126.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.126.4. Decode Variables

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

### C.126.5. IDL Operation

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

### C.126.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.126.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

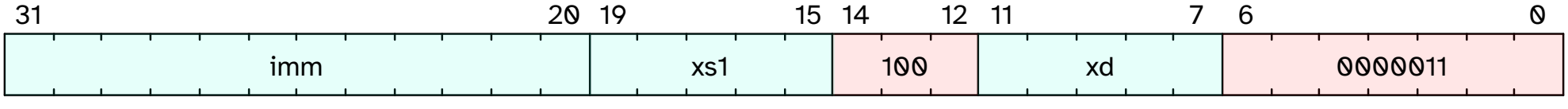
## C.127. lbu

### Load byte unsigned

This instruction is defined by:

- I, version >= I@2.1.0

### C.127.1. Encoding




### C.127.2. Description

Load 8 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Zero extend the result.

### C.127.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.127.4. Decode Variables

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

### C.127.5. IDL Operation

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

### C.127.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.127.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

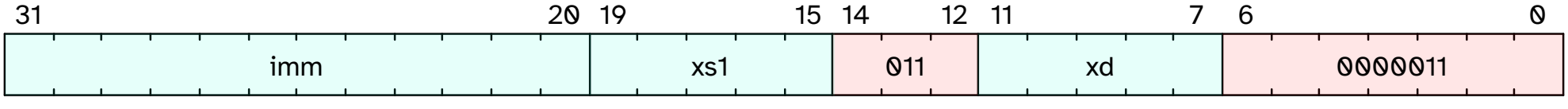
## C.128. ld

**Load doubleword**

This instruction is defined by:

- I, version >= I@2.1.0

### C.128.1. Encoding




### C.128.2. Description

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

### C.128.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.128.4. Decode Variables

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

### C.128.5. IDL Operation

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

### C.128.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.128.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

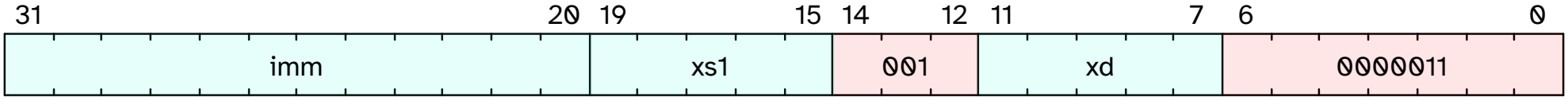
## C.129. lh

**Load halfword**

This instruction is defined by:

- I, version >= I@2.1.0

### C.129.1. Encoding




### C.129.2. Description

Load 16 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Sign extend the result.

### C.129.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.129.4. Decode Variables

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

### C.129.5. IDL Operation

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

### C.129.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.129.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

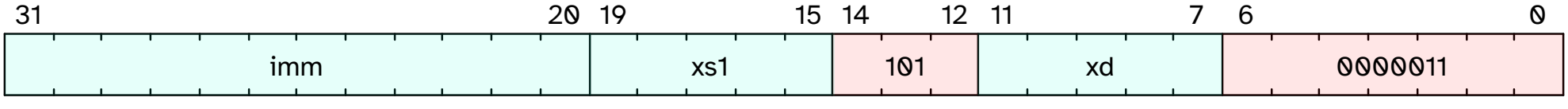
## C.130. lhu

**Load halfword unsigned**

This instruction is defined by:

- I, version >= I@2.1.0

### C.130.1. Encoding




### C.130.2. Description

Load 16 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Zero extend the result.

### C.130.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.130.4. Decode Variables

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

### C.130.5. IDL Operation

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

### C.130.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.130.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

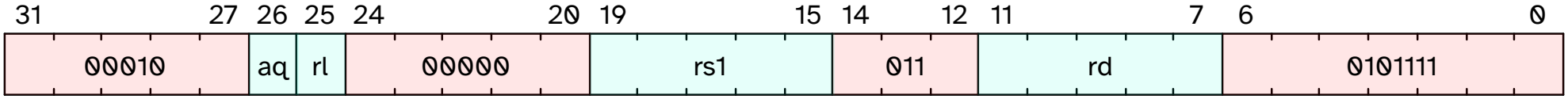
## C.131. lr.d

**Load reserved doubleword**

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

### C.131.1. Encoding




### C.131.2. Description

Loads a word from the address in rs1, places the value in rd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

The address in rs1 must be 8-byte aligned.

If the address is not naturally aligned, a `LoadAddressMisaligned` exception or an `LoadAccessFault` exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

### C.131.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.131.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.131.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address
is misaligned");
```

```
  }
}
X[rd] = load_reserved<32>(virtual_address, aq, rl, $encoding);
```

### C.131.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Extensions might perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), Read(Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        let aligned : bool =
           /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
            * to treat them as valid here; otherwise we'd need to throw an internal_error.
            */
           match width {
             BYTE   => true,
             HALF   => vaddr[0..0] == 0b0,
             WORD   => vaddr[1..0] == 0b00,
             DOUBLE => vaddr[2..0] == 0b000
           };
        /* "LR faults like a normal load, even though it's in the AMO major opcode space."
         * - Andrew Waterman, isa-dev, 10 Jul 2018.
         */
        if not(aligned)
        then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
        else match translateAddr(vaddr, Read(Data)) {
               TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
               TR_Address(addr, _) =>
                 match (width, sizeof(xlen)) {
                   (BYTE, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 1, aq, aq & rl, true), false),
                   (HALF, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 2, aq, aq & rl, true), false),
                   (WORD, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 4, aq, aq & rl, true), false),
                   (DOUBLE, 64) => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 8, aq, aq & rl, true), false),
                   _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
                 }
             }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

### C.131.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

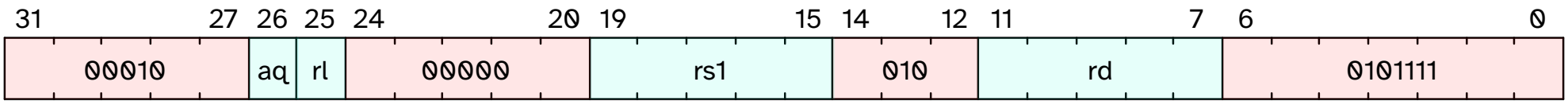
## C.132. lr.w

**Load reserved word**

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

### C.132.1. Encoding




### C.132.2. Description

Loads a word from the address in rs1, places the sign-extended value in rd, and registers a *reservation set* — a set of bytes that subsumes the bytes in the addressed word.

<%- if MXLEN == 64 -%> The 32-bit load result is sign-extended to 64-bits. <%- end -%>

The address in rs1 must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words).

If the address is not naturally aligned, a `LoadAddressMisaligned` exception or an `LoadAccessFault` exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

### C.132.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.132.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.132.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
```

```
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address
is misaligned");
  }
}
XReg load_value = load_reserved<32>(virtual_address, aq, rl, $encoding);
if (xlen() == 64) {
  X[rd] = load_value;
} else {
  X[rd] = sext(load_value[31:0], 32);
}
```

### C.132.6. Sail Operation

```
{
  if extension("A") then {
    /* Get the address, X(rs1) (no offset).
     * Extensions might perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), Read(Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        let aligned : bool =
        /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
         * to treat them as valid here; otherwise we'd need to throw an internal_error.
         */
        match width {
          BYTE   => true,
          HALF   => vaddr[0..0] == 0b0,
          WORD   => vaddr[1..0] == 0b00,
          DOUBLE => vaddr[2..0] == 0b000
        };
    /* "LR faults like a normal load, even though it's in the AMO major opcode space."
     * - Andrew Waterman, isa-dev, 10 Jul 2018.
     */
    if not(aligned)
    then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
    else match translateAddr(vaddr, Read(Data)) {
           TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
           TR_Address(addr, _) =>
             match (width, sizeof(xlen)) {
               (BYTE, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 1, aq, aq & rl, true), false),
               (HALF, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 2, aq, aq & rl, true), false),
               (WORD, _)    => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 4, aq, aq & rl, true), false),
               (DOUBLE, 64) => process_loadres(rd, vaddr, mem_read(Read(Data), addr, 8, aq, aq & rl, true), false),
               _            => internal_error(__FILE__, __LINE__, "Unexpected AMO width")
             }
         }
  }
} else {
  handle_illegal();
  RETIRE_FAIL
}
}
```

### C.132.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

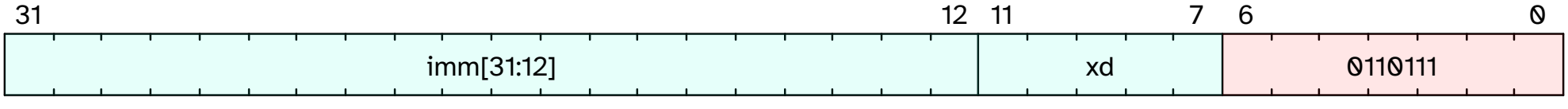
## C.133. lui

### Load upper immediate

This instruction is defined by:

- I, version >= I@2.1.0

### C.133.1. Encoding




### C.133.2. Description

Load the zero-extended imm into xd.

### C.133.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.133.4. Decode Variables

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

### C.133.5. IDL Operation

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

### C.133.6. Sail Operation

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

### C.133.7. Exceptions

This instruction does not generate synchronous exceptions.

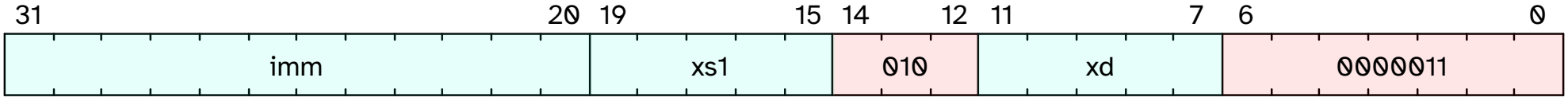
## C.134. lw

**Load word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.134.1. Encoding




### C.134.2. Description

Load 32 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset. Sign extend the result.

### C.134.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.134.4. Decode Variables

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

### C.134.5. IDL Operation

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

### C.134.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.134.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

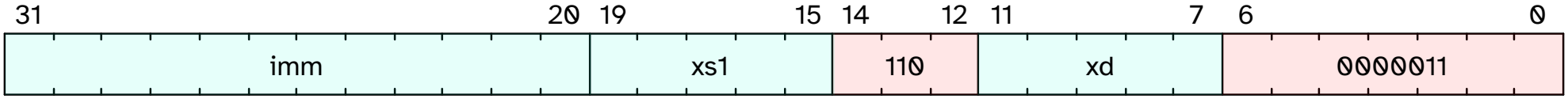
## C.135. lwu

### Load word unsigned

This instruction is defined by:

- I, version >= I@2.1.0

### C.135.1. Encoding




### C.135.2. Description

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

### C.135.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.135.4. Decode Variables

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

### C.135.5. IDL Operation

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

### C.135.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Read(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
      then { handle_mem_exception(vaddr, E_Load_Addr_Align()); RETIRE_FAIL }
      else match translateAddr(vaddr, Read(Data)) {
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(xd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

### C.135.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

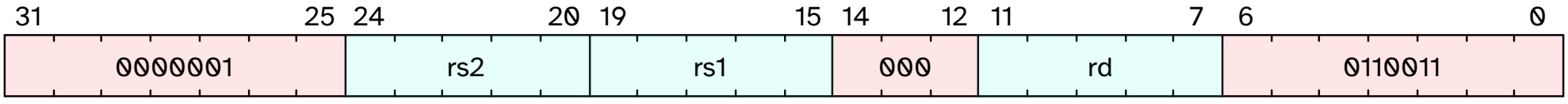
## C.136. mul

**Signed multiply**

This instruction is defined by:

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

### C.136.1. Encoding




### C.136.2. Description

MUL performs an XLEN-bitxXLEN-bit multiplication of `rs1` by `rs2` and places the lower XLEN bits in the destination register. Any overflow is thrown away.

*If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1, rs2 (source register specifiers must be in same order and rdh cannot be the same as rs1 or rs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.*

### C.136.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.136.4. Decode Variables

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

### C.136.5. IDL Operation

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

### C.136.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

### C.136.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

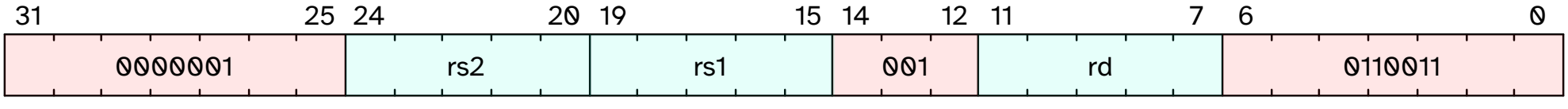
## C.137. mulh

### Signed multiply high

This instruction is defined by:

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

### C.137.1. Encoding




### C.137.2. Description

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

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

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

Microarchitectures may look for that sequence and fuse the operations.

### C.137.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.137.4. Decode Variables

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

### C.137.5. IDL Operation

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

### C.137.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
```

```
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

### C.137.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

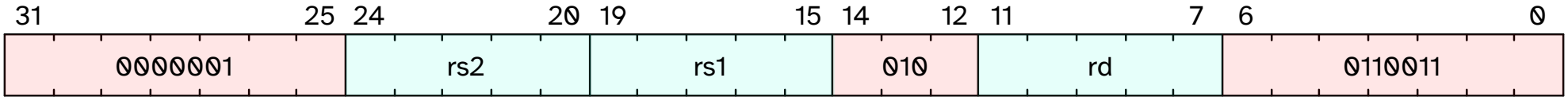
## C.138. mulhsu

**Signed/unsigned multiply high**

This instruction is defined by:

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

### C.138.1. Encoding




### C.138.2. Description

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

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

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

Microarchitectures may look for that sequence and fuse the operations.

### C.138.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.138.4. Decode Variables

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

### C.138.5. IDL Operation

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

### C.138.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
```

```
    handle_illegal();
    RETIRE_FAIL
  }
}
```

### C.138.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

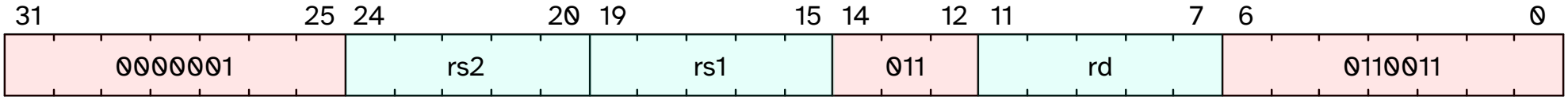
## C.139. mulhu

### Unsigned multiply high

This instruction is defined by:

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

### C.139.1. Encoding




### C.139.2. Description

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

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

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

Microarchitectures may look for that sequence and fuse the operations.

### C.139.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.139.4. Decode Variables

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

### C.139.5. IDL Operation

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

### C.139.6. Sail Operation

```
{
  if extension("M") | haveZmmul() then {
    let rs1_val = X(rs1);
    let rs2_val = X(rs2);
    let rs1_int : int = if signed1 then signed(rs1_val) else unsigned(rs1_val);
    let rs2_int : int = if signed2 then signed(rs2_val) else unsigned(rs2_val);
    let result_wide = to_bits(2 * sizeof(xlen), rs1_int * rs2_int);
    let result = if   high
                 then result_wide[(2 * sizeof(xlen) - 1) .. sizeof(xlen)]
                 else result_wide[(sizeof(xlen) - 1) .. 0];
    X(rd) = result;
    RETIRE_SUCCESS
  } else {
    handle_illegal();
```

```
    RETIRE_FAIL
  }
}
```

### C.139.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

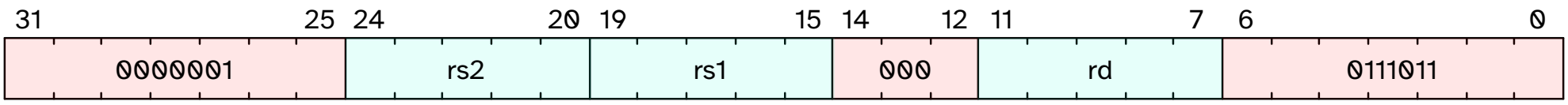
## C.140. mulw

**Signed 32-bit multiply**

This instruction is defined by:

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

### C.140.1. Encoding




### C.140.2. Description

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

Any overflow is thrown away.

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

### C.140.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.140.4. Decode Variables

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

### C.140.5. IDL Operation

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

### C.140.6. Sail Operation

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

### C.140.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

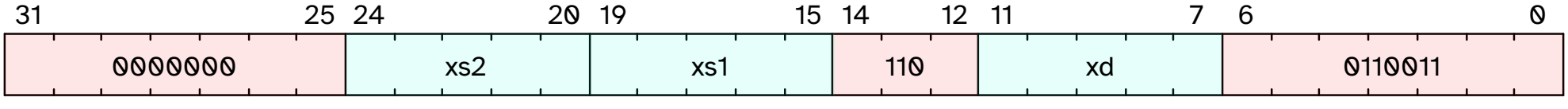
## C.141. or

**Or**

This instruction is defined by:

- I, version >= I@2.1.0

### C.141.1. Encoding




### C.141.2. Description

Or xs1 with xs2, and store the result in xd

### C.141.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.141.4. Decode Variables

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

### C.141.5. IDL Operation

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

### C.141.6. Sail Operation

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

### C.141.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.142. ori

### Or immediate

This instruction is defined by:

- I, version >= I@2.1.0

### C.142.1. Encoding


### C.142.2. Description

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

### C.142.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.142.4. Decode Variables

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

### C.142.5. IDL Operation

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

### C.142.6. Sail Operation

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

### C.142.7. Exceptions

This instruction does not generate synchronous exceptions.

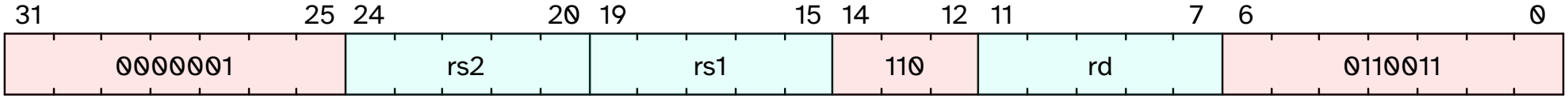
## C.143. rem

### Signed remainder

This instruction is defined by:

- M, version >= M@2.0.0

### C.143.1. Encoding




### C.143.2. Description

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

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

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

### C.143.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.143.4. Decode Variables

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

### C.143.5. IDL Operation

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

### C.143.6. Sail Operation

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

### C.143.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

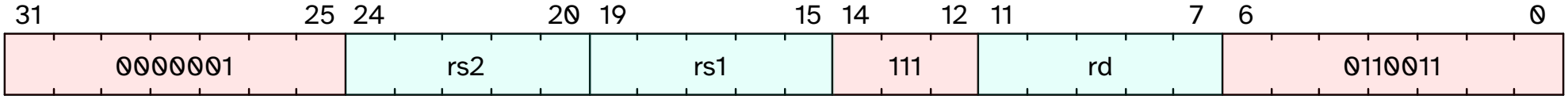
## C.144. remu

### Unsigned remainder

This instruction is defined by:

- M, version >= M@2.0.0

### C.144.1. Encoding




### C.144.2. Description

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

### C.144.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.144.4. Decode Variables

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

### C.144.5. IDL Operation

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

### C.144.6. Sail Operation

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

### C.144.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

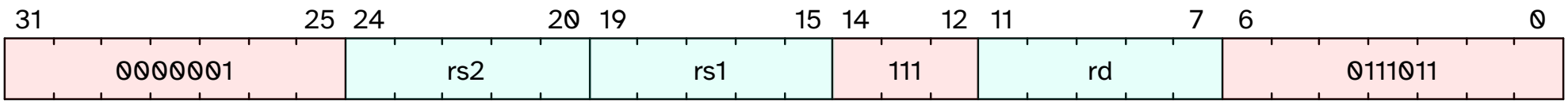
## C.145. remuw

**Unsigned 32-bit remainder**

This instruction is defined by:

- M, version >= M@2.0.0

### C.145.1. Encoding




### C.145.2. Description

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

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

### C.145.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.145.4. Decode Variables

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

### C.145.5. IDL Operation

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

### C.145.6. Sail Operation

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

### C.145.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

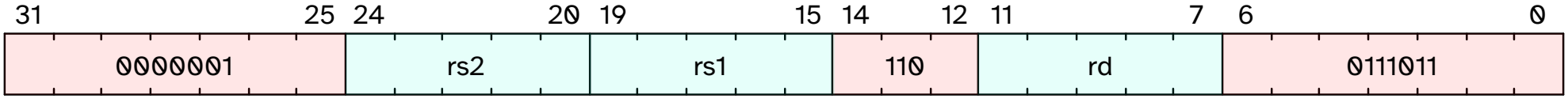
## C.146. remw

**Signed 32-bit remainder**

This instruction is defined by:

- M, version >= M@2.0.0

### C.146.1. Encoding




### C.146.2. Description

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

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

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

### C.146.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.146.4. Decode Variables

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

### C.146.5. IDL Operation

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

### C.146.6. Sail Operation

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

```
}
```

### C.146.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

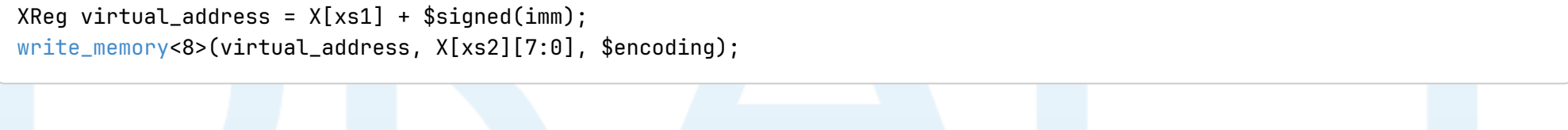
## C.147. sb

**Store byte**

This instruction is defined by:

- I, version >= I@2.1.0

### C.147.1. Encoding




### C.147.2. Description

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

### C.147.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.147.4. Decode Variables

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

### C.147.5. IDL Operation

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

### C.147.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
  then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
  else match translateAddr(vaddr, Write(Data)) {
    TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
    TR_Address(paddr, _) => {
      let eares : MemoryOpResult(unit) = match width {
        BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
        HALF   => mem_write_ea(paddr, 2, aq, rl, false),
        WORD   => mem_write_ea(paddr, 4, aq, rl, false),
        DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let xs2_val = X(xs2);
          let res : MemoryOpResult(bool) = match (width) {
            BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
            HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
            WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
            DOUBLE if sizeof(xlen) >= 64
                 => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
            _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
          };
          match (res) {
```

```
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
  }
}
```

### C.147.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

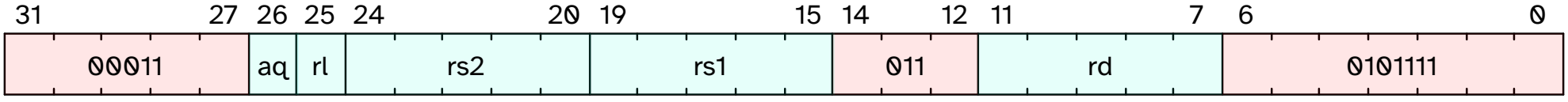
## C.148. sc.d

**Store conditional doubleword**

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

### C.148.1. Encoding




### C.148.2. Description

sc.d conditionally writes a doubleword in *rs2* to the address in *rs1*: the sc.d succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.d succeeds, the instruction writes the doubleword in *rs2* to memory, and it writes zero to *rd*. If the sc.d fails, the instruction does not write to memory, and it writes a nonzero value to *rd*. For the purposes of memory protection, a failed sc.d may be treated like a store. Regardless of success or failure, executing an sc.d instruction invalidates any reservation held by this hart.

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in *rs1* must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

> *Emulating misaligned LR/SC sequences is impractical in most systems.*
>
> *Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.*

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

> *The platform should provide a means to determine the size and shape of the reservation set.*
>
> *A platform specification may constrain the size and shape of the reservation set.*
>
> *A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:*
>
> - *during a preemptive context switch, and*
> - *if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.*
>
> *The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.*

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

> *The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and*

> *after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.*
>
> *If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.*

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

### C.148.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.148.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.148.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<64>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address
is misaligned");
  }
}
Boolean success = store_conditional<64>(virtual_address, value, aq, rl, $encoding);
X[rd] = success ? 0 : 1;
```

### C.148.6. Sail Operation

```
{
if speculate_conditional () == false then {
  /* should only happen in rmem
   * rmem: allow SC to fail very early
   */
  X(rd) = zero_extend(0b1); RETIRE_SUCCESS
} else {
  if extension("A") then {
    /* normal non-rmem case
     * rmem: SC is allowed to succeed (but might fail later)
     */
    /* Get the address, X(rs1) (no offset).
     * Extensions might perform additional checks on address validity.
     */
    match ext_data_get_addr(rs1, zeros(), Write(Data), width) {
      Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
      Ext_DataAddr_OK(vaddr) => {
        let aligned : bool =
          /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
           * to treat them as valid here; otherwise we'd need to throw an internal_error.
           */
          match width {
            BYTE   => true,
            HALF   => vaddr[0..0] == 0b0,
            WORD   => vaddr[1..0] == 0b00,
```

```
      DOUBLE => vaddr[2..0] == 0b000
    };
  if not(aligned)
  then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
  else {
    if match_reservation(vaddr) == false then {
      /* cannot happen in rmem */
      X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS
    } else {
      match translateAddr(vaddr, Write(Data)) {  /* Write and ReadWrite are equivalent here:
                                                  * both result in a SAMO exception */
        TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        TR_Address(addr, _) => {
          let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
            (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
            (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
            (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
            (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
            _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
          };
          match (eares) {
            MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
            MemValue(_) => {
              rs2_val = X(rs2);
              let res : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                (BYTE, _)    => mem_write_value(addr, 1, rs2_val[7..0], aq & rl, rl, true),
                (HALF, _)    => mem_write_value(addr, 2, rs2_val[15..0], aq & rl, rl, true),
                (WORD, _)    => mem_write_value(addr, 4, rs2_val[31..0], aq & rl, rl, true),
                (DOUBLE, 64) => mem_write_value(addr, 8, rs2_val,        aq & rl, rl, true),
                _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
              };
              match (res) {
                MemValue(true)  => { X(rd) = zero_extend(0b0); cancel_reservation(); RETIRE_SUCCESS },
                MemValue(false) => { X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS },
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
    } else {
      handle_illegal();
      RETIRE_FAIL
    }
  }
}
```

### C.148.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

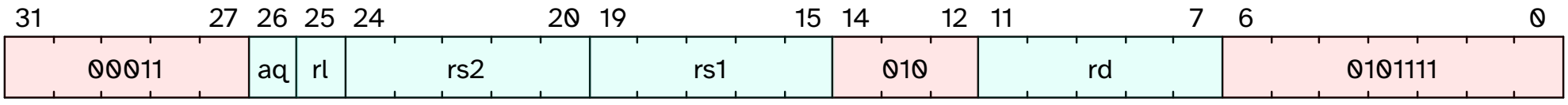
## C.149. sc.w

### Store conditional word

This instruction is defined by:

- Zalrsc, version >= Zalrsc@1.0.0

### C.149.1. Encoding




### C.149.2. Description

sc.w conditionally writes a word in *rs2* to the address in *rs1*: the sc.w succeeds only if the reservation is still valid and the reservation set contains the bytes being written. If the sc.w succeeds, the instruction writes the word in *rs2* to memory, and it writes zero to *rd*. If the sc.w fails, the instruction does not write to memory, and it writes a nonzero value to *rd*. For the purposes of memory protection, a failed sc.w may be treated like a store. Regardless of success or failure, executing an sc.w instruction invalidates any reservation held by this hart.

<%- if MXLEN == 64 -%>

*If a value other than 0 or 1 is defined as a result for sc.w, the value will before sign-extended into rd. <%- end -%>*

The failure code with value 1 encodes an unspecified failure. Other failure codes are reserved at this time. Portable software should only assume the failure code will be non-zero.

The address held in *rs1* must be naturally aligned to the size of the operand (*i.e.*, eight-byte aligned for doublewords and four-byte aligned for words). If the address is not naturally aligned, an address-misaligned exception or an access-fault exception will be generated. The access-fault exception can be generated for a memory access that would otherwise be able to complete except for the misalignment, if the misaligned access should not be emulated.

*Emulating misaligned LR/SC sequences is impractical in most systems.*

*Misaligned LR/SC sequences also raise the possibility of accessing multiple reservation sets at once, which present definitions do not provide for.*

An implementation can register an arbitrarily large reservation set on each LR, provided the reservation set includes all bytes of the addressed data word or doubleword. An SC can only pair with the most recent LR in program order. An SC may succeed only if no store from another hart to the reservation set can be observed to have occurred between the LR and the SC, and if there is no other SC between the LR and itself in program order. An SC may succeed only if no write from a device other than a hart to the bytes accessed by the LR instruction can be observed to have occurred between the LR and SC. Note this LR might have had a different effective address and data size, but reserved the SC's address as part of the reservation set.

```
Following this model, in systems with memory translation, an SC is allowed to succeed if the
earlier LR reserved the same location using an alias with a different virtual address, but is
also allowed to fail if the virtual address is different.

To accommodate legacy devices and buses, writes from devices other than RISC-V harts are only
required to invalidate reservations when they overlap the bytes accessed by the LR.
These writes are not required to invalidate the reservation when they access other bytes in
the reservation set.
```

The SC must fail if the address is not within the reservation set of the most recent LR in program order. The SC must fail if a store to the reservation set from another hart can be observed to occur between the LR and SC. The SC must fail if a write from some other device to the bytes accessed by the LR can be observed to occur between the LR and SC. (If such a device writes the reservation set but does not write the bytes accessed by the LR, the SC may or may not fail.) An SC must fail if there is another SC (to any address) between the LR and the SC in program order. The precise statement of the atomicity requirements for successful LR/SC sequences is defined by the Atomicity Axiom of the memory model.

*The platform should provide a means to determine the size and shape of the reservation set.*

*A platform specification may constrain the size and shape of the reservation set.*

*A store-conditional instruction to a scratch word of memory should be used to forcibly invalidate any existing load reservation:*

- *during a preemptive context switch, and*
- *if necessary when changing virtual to physical address mappings, such as when migrating pages that might contain an active reservation.*

*The invalidation of a hart's reservation when it executes an LR or SC imply that a hart can only hold one reservation at a time, and that an SC can only pair with the most recent LR, and LR with the next following SC, in program order. This is a restriction to the Atomicity Axiom in Section 18.1 that ensures software runs correctly on expected common implementations that operate in this manner.*

An SC instruction can never be observed by another RISC-V hart before the LR instruction that established the reservation.

*The LR/SC sequence can be given acquire semantics by setting the aq bit on the LR instruction. The LR/SC sequence can be given*

> *release semantics by by setting the rl bit on the SC instruction. Assuming suitable mappings for other atomic operations, setting the aq bit on the LR instruction, and setting the rl bit on the SC instruction makes the LR/SC sequence sequentially consistent in the C memory_order_seq_cst sense. Such a sequence does not act as a fence for ordering ordinary load and store instructions before and after the sequence. Specific instruction mappings for other C atomic operations, or stronger notions of "sequential consistency", may require both bits to be set on either or both of the LR or SC instruction.*
>
> *If neither bit is set on either LR or SC, the LR/SC sequence can be observed to occur before or after surrounding memory operations from the same RISC-V hart. This can be appropriate when the LR/SC sequence is used to implement a parallel reduction operation.*

Software should not set the *rl* bit on an LR instruction unless the *aq* bit is also set. LR.rl and SC.aq instructions are not guaranteed to provide any stronger ordering than those with both bits clear, but may result in lower performance.

### C.149.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.149.4. Decode Variables

```
Bits<1> aq = $encoding[26];
Bits<1> rl = $encoding[25];
Bits<5> rs2 = $encoding[24:20];
Bits<5> rs1 = $encoding[19:15];
Bits<5> rd = $encoding[11:7];
```

### C.149.5. IDL Operation

```
if (implemented?(ExtensionName::A) && (misa.A == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[rs1];
XReg value = X[rs2];
if (!is_naturally_aligned<32>(virtual_address)) {
  if (LRSC_MISALIGNED_BEHAVIOR == "always raise misaligned exception") {
    raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
  } else if (LRSC_MISALIGNED_BEHAVIOR == "always raise access fault") {
    raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
  } else {
    unpredictable("Implementations may raise either a LoadAddressMisaligned or a LoadAccessFault when an LR/SC address
is misaligned");
  }
}
Boolean success = store_conditional<32>(virtual_address, value, aq, rl, $encoding);
X[rd] = success ? 0 : 1;
```

### C.149.6. Sail Operation

```
{
  if speculate_conditional () == false then {
    /* should only happen in rmem
     * rmem: allow SC to fail very early
     */
    X(rd) = zero_extend(0b1); RETIRE_SUCCESS
  } else {
    if extension("A") then {
      /* normal non-rmem case
       * rmem: SC is allowed to succeed (but might fail later)
       */
      /* Get the address, X(rs1) (no offset).
       * Extensions might perform additional checks on address validity.
       */
      match ext_data_get_addr(rs1, zeros(), Write(Data), width) {
        Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
        Ext_DataAddr_OK(vaddr) => {
          let aligned : bool =
            /* BYTE and HALF would only occur due to invalid decodes, but it doesn't hurt
             * to treat them as valid here; otherwise we'd need to throw an internal_error.
             */
            match width {
```

```
            BYTE   => true,
            HALF   => vaddr[0..0] == 0b0,
            WORD   => vaddr[1..0] == 0b00,
            DOUBLE => vaddr[2..0] == 0b000
          };
        if not(aligned)
        then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
        else {
          if match_reservation(vaddr) == false then {
            /* cannot happen in rmem */
            X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS
          } else {
            match translateAddr(vaddr, Write(Data)) {  /* Write and ReadWrite are equivalent here:
                                                        * both result in a SAMO exception */
              TR_Failure(e, _) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
              TR_Address(addr, _) => {
                let eares : MemoryOpResult(unit) = match (width, sizeof(xlen)) {
                  (BYTE, _)    => mem_write_ea(addr, 1, aq & rl, rl, true),
                  (HALF, _)    => mem_write_ea(addr, 2, aq & rl, rl, true),
                  (WORD, _)    => mem_write_ea(addr, 4, aq & rl, rl, true),
                  (DOUBLE, 64) => mem_write_ea(addr, 8, aq & rl, rl, true),
                  _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
                };
                match (eares) {
                  MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
                  MemValue(_) => {
                    rs2_val = X(rs2);
                    let res : MemoryOpResult(bool) = match (width, sizeof(xlen)) {
                      (BYTE, _)    => mem_write_value(addr, 1, rs2_val[7..0], aq & rl, rl, true),
                      (HALF, _)    => mem_write_value(addr, 2, rs2_val[15..0], aq & rl, rl, true),
                      (WORD, _)    => mem_write_value(addr, 4, rs2_val[31..0], aq & rl, rl, true),
                      (DOUBLE, 64) => mem_write_value(addr, 8, rs2_val,        aq & rl, rl, true),
                      _            => internal_error(__FILE__, __LINE__, "STORECON expected word or double")
                    };
                    match (res) {
                      MemValue(true)  => { X(rd) = zero_extend(0b0); cancel_reservation(); RETIRE_SUCCESS },
                      MemValue(false) => { X(rd) = zero_extend(0b1); cancel_reservation(); RETIRE_SUCCESS },
                      MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  } else {
    handle_illegal();
    RETIRE_FAIL
  }
}
```

### C.149.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- StoreAmoAccessFault
- StoreAmoPageFault

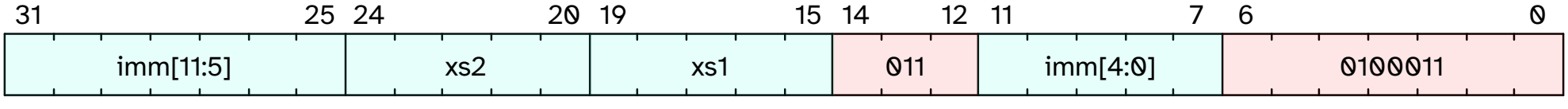
## C.150. sd

**Store doubleword**

This instruction is defined by:

- I, version >= I@2.1.0

### C.150.1. Encoding




### C.150.2. Description

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

### C.150.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.150.4. Decode Variables

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

### C.150.5. IDL Operation

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

### C.150.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
  then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
  else match translateAddr(vaddr, Write(Data)) {
    TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
    TR_Address(paddr, _) => {
      let eares : MemoryOpResult(unit) = match width {
        BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
        HALF   => mem_write_ea(paddr, 2, aq, rl, false),
        WORD   => mem_write_ea(paddr, 4, aq, rl, false),
        DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let xs2_val = X(xs2);
          let res : MemoryOpResult(bool) = match (width) {
            BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
            HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
            WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
            DOUBLE if sizeof(xlen) >= 64
                 => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
            _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
          };
          match (res) {
```

```
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
  }
}
```

### C.150.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

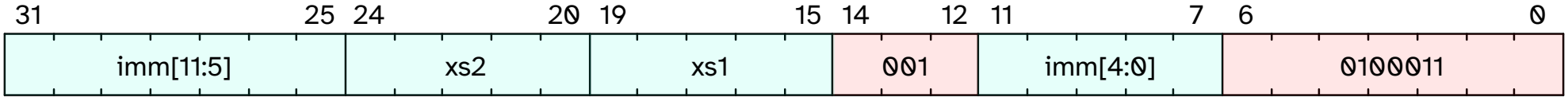
## C.151. sh

**Store halfword**

This instruction is defined by:

- I, version >= I@2.1.0

### C.151.1. Encoding




### C.151.2. Description

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

### C.151.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.151.4. Decode Variables

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

### C.151.5. IDL Operation

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

### C.151.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
  then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
  else match translateAddr(vaddr, Write(Data)) {
    TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
    TR_Address(paddr, _) => {
      let eares : MemoryOpResult(unit) = match width {
        BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
        HALF   => mem_write_ea(paddr, 2, aq, rl, false),
        WORD   => mem_write_ea(paddr, 4, aq, rl, false),
        DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let xs2_val = X(xs2);
          let res : MemoryOpResult(bool) = match (width) {
            BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
            HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
            WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
            DOUBLE if sizeof(xlen) >= 64
                 => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
            _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
          };
          match (res) {
```

```
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
  }
}
```

### C.151.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

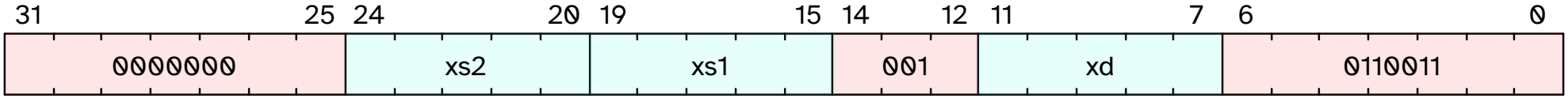
## C.152. sll

**Shift left logical**

This instruction is defined by:

- I, version >= I@2.1.0

### C.152.1. Encoding




### C.152.2. Description

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

### C.152.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.152.4. Decode Variables

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

### C.152.5. IDL Operation

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

### C.152.6. Sail Operation

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

### C.152.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.153. slli

### Shift left logical immediate

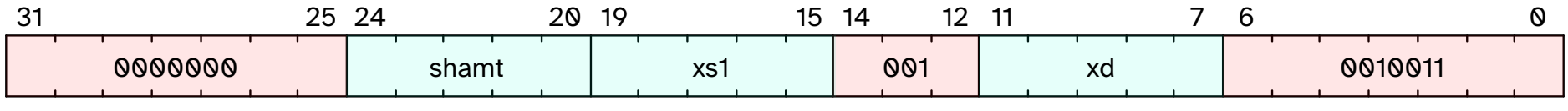
This instruction is defined by:

- I, version >= I@2.1.0

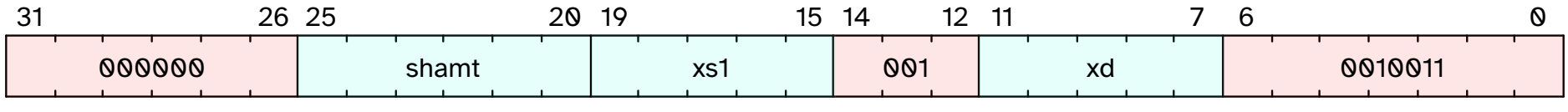
### C.153.1. Encoding

*This instruction has different encodings in RV32 and RV64.*

**RV32**




**RV64**




### C.153.2. Description

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

### C.153.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.153.4. Decode Variables

**RV32**

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

**RV64**

```
Bits<6> shamt = $encoding[25:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

### C.153.5. IDL Operation

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

### C.153.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then xs1_val << shamt[4..0]
                  else xs1_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then xs1_val >> shamt[4..0]
                  else xs1_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, shamt[4..0])
                  else shift_right_arith64(xs1_val, shamt)
```

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

### C.153.7. Exceptions

This instruction does not generate synchronous exceptions.

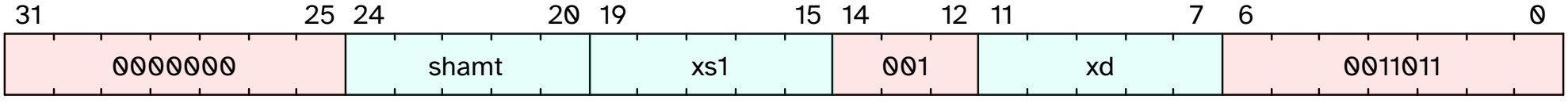
## C.154. slliw

**Shift left logical immediate word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.154.1. Encoding




### C.154.2. Description

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

### C.154.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.154.4. Decode Variables

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

### C.154.5. IDL Operation

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

### C.154.6. Sail Operation

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

### C.154.7. Exceptions

This instruction does not generate synchronous exceptions.

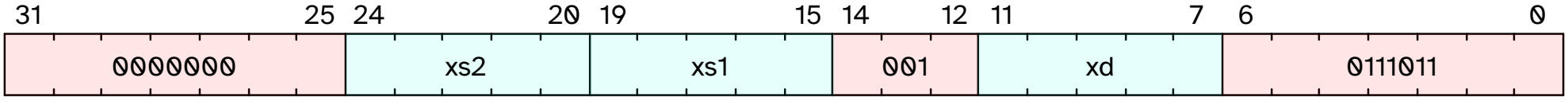
## C.155. sllw

**Shift left logical word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.155.1. Encoding




### C.155.2. Description

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

### C.155.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.155.4. Decode Variables

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

### C.155.5. IDL Operation

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

### C.155.6. Sail Operation

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

### C.155.7. Exceptions

This instruction does not generate synchronous exceptions.

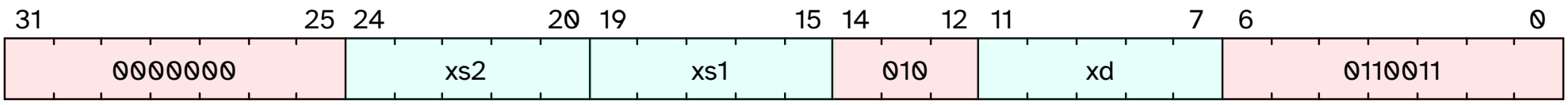
## C.156. slt

**Set on less than**

This instruction is defined by:

- I, version >= I@2.1.0

### C.156.1. Encoding




### C.156.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the value in register `xs2`, where both sources are treated as signed numbers, else `0` is written to `xd`.

### C.156.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.156.4. Decode Variables

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

### C.156.5. IDL Operation

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

### C.156.6. Sail Operation

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

### C.156.7. Exceptions

This instruction does not generate synchronous exceptions.

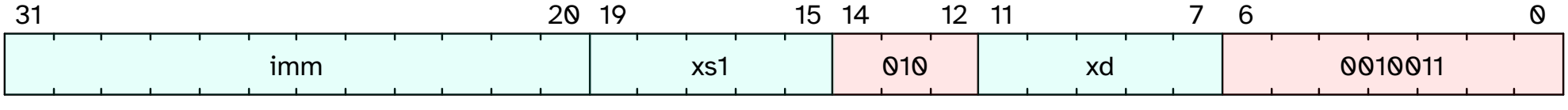
## C.157. slti

**Set on less than immediate**

This instruction is defined by:

- I, version >= I@2.1.0

### C.157.1. Encoding




### C.157.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to `xd`.

### C.157.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.157.4. Decode Variables

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

### C.157.5. IDL Operation

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

### C.157.6. Sail Operation

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

### C.157.7. Exceptions

This instruction does not generate synchronous exceptions.

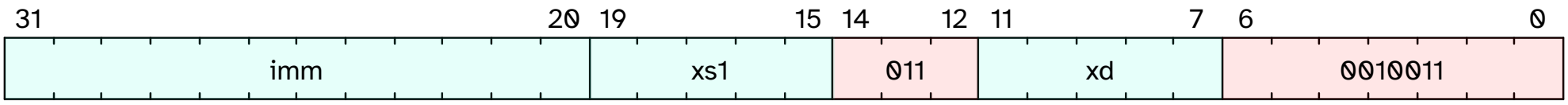
## C.158. sltiu

**Set on less than immediate unsigned**

This instruction is defined by:

- I, version >= I@2.1.0

### C.158.1. Encoding




### C.158.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the sign-extended immediate when both are treated as unsigned numbers (*i.e.*, the immediate is first sign-extended to XLEN bits then treated as an unsigned number), else 0 is written to `xd`.

`sltiu xd, xs1, 1` *sets* `xd` *to 1 if* `xs1` *equals zero, otherwise sets* `xd` *to 0 (assembler pseudoinstruction* `SEQZ xd, rs`*).*

### C.158.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.158.4. Decode Variables

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

### C.158.5. IDL Operation

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

### C.158.6. Sail Operation

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

### C.158.7. Exceptions

This instruction does not generate synchronous exceptions.

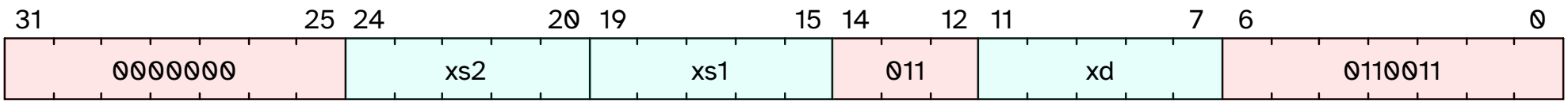
## C.159. sltu

**Set on less than unsigned**

This instruction is defined by:

- I, version >= I@2.1.0

### C.159.1. Encoding




### C.159.2. Description

Places the value 1 in register `xd` if register `xs1` is less than the value in register `xs2`, where both sources are treated as unsigned numbers, else `0` is written to `xd`.

### C.159.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.159.4. Decode Variables

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

### C.159.5. IDL Operation

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

### C.159.6. Sail Operation

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

### C.159.7. Exceptions

This instruction does not generate synchronous exceptions.

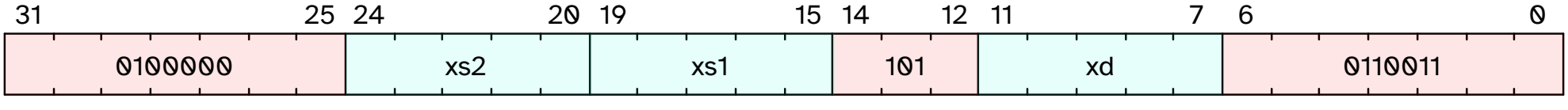
## C.160. sra

**Shift right arithmetic**

This instruction is defined by:

- I, version >= I@2.1.0

### C.160.1. Encoding




### C.160.2. Description

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

### C.160.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.160.4. Decode Variables

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

### C.160.5. IDL Operation

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

### C.160.6. Sail Operation

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

### C.160.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.161. srai

### Shift right arithmetic immediate

This instruction is defined by:

- I, version >= I@2.1.0

### C.161.1. Encoding

*This instruction has different encodings in RV32 and RV64.*

**RV32**


**RV64**


### C.161.2. Description

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

### C.161.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.161.4. Decode Variables

**RV32**

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

**RV64**

```
Bits<6> shamt = $encoding[25:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

### C.161.5. IDL Operation

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

### C.161.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then xs1_val << shamt[4..0]
                  else xs1_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then xs1_val >> shamt[4..0]
                  else xs1_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, shamt[4..0])
                  else shift_right_arith64(xs1_val, shamt)
```

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

### C.161.7. Exceptions

This instruction does not generate synchronous exceptions.

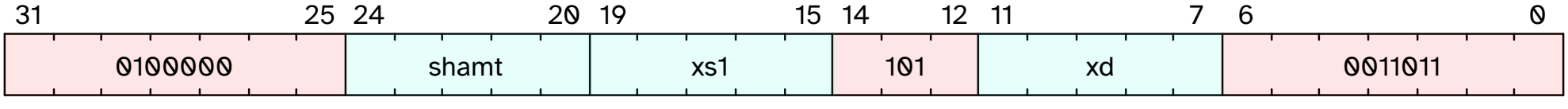
## C.162. sraiw

**Shift right arithmetic immediate word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.162.1. Encoding




### C.162.2. Description

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

### C.162.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.162.4. Decode Variables

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

### C.162.5. IDL Operation

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

### C.162.6. Sail Operation

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

### C.162.7. Exceptions

This instruction does not generate synchronous exceptions.

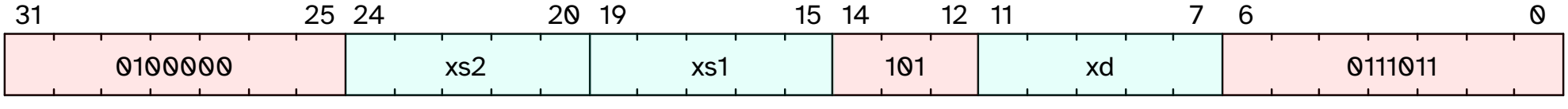
## C.163. sraw

**Shift right arithmetic word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.163.1. Encoding




### C.163.2. Description

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

### C.163.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.163.4. Decode Variables

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

### C.163.5. IDL Operation

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

### C.163.6. Sail Operation

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

### C.163.7. Exceptions

This instruction does not generate synchronous exceptions.

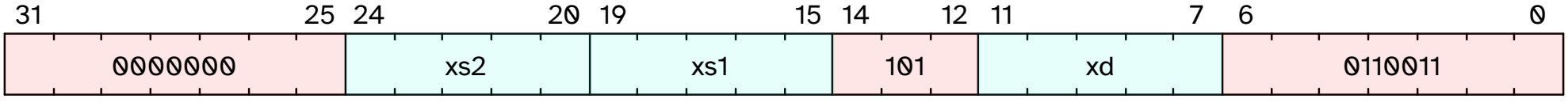
## C.164. srl

**Shift right logical**

This instruction is defined by:

- I, version >= I@2.1.0

### C.164.1. Encoding




### C.164.2. Description

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

### C.164.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.164.4. Decode Variables

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

### C.164.5. IDL Operation

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

### C.164.6. Sail Operation

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

### C.164.7. Exceptions

This instruction does not generate synchronous exceptions.

## C.165. srli

### Shift right logical immediate

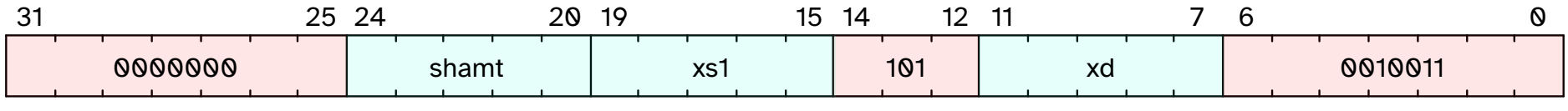
This instruction is defined by:

- I, version >= I@2.1.0

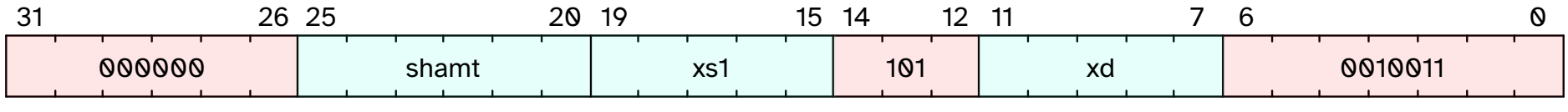
### C.165.1. Encoding

*This instruction has different encodings in RV32 and RV64.*

**RV32**




**RV64**




### C.165.2. Description

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

### C.165.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.165.4. Decode Variables

**RV32**

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

**RV64**

```
Bits<6> shamt = $encoding[25:20];
Bits<5> xs1 = $encoding[19:15];
Bits<5> xd = $encoding[11:7];
```

### C.165.5. IDL Operation

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

### C.165.6. Sail Operation

```
{
  let xs1_val = X(xs1);
  /* the decoder guaxd should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then xs1_val << shamt[4..0]
                  else xs1_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then xs1_val >> shamt[4..0]
                  else xs1_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(xs1_val, shamt[4..0])
                  else shift_right_arith64(xs1_val, shamt)
```

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

### C.165.7. Exceptions

This instruction does not generate synchronous exceptions.

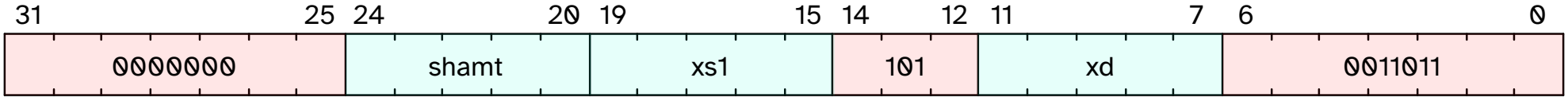
## C.166. srliw

**Shift right logical immediate word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.166.1. Encoding




### C.166.2. Description

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

### C.166.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.166.4. Decode Variables

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

### C.166.5. IDL Operation

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

### C.166.6. Sail Operation

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

### C.166.7. Exceptions

This instruction does not generate synchronous exceptions.

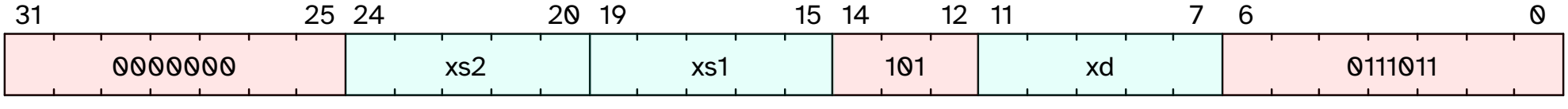
## C.167. srlw

**Shift right logical word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.167.1. Encoding




### C.167.2. Description

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

### C.167.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.167.4. Decode Variables

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

### C.167.5. IDL Operation

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

### C.167.6. Sail Operation

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

### C.167.7. Exceptions

This instruction does not generate synchronous exceptions.

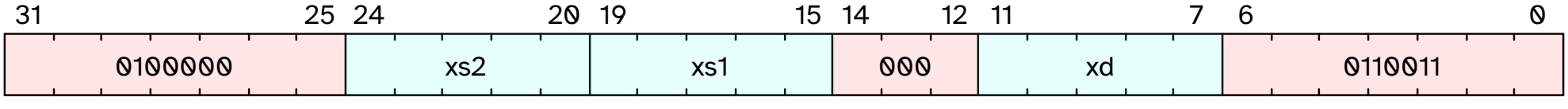
## C.168. sub

**Subtract**

This instruction is defined by:

- I, version >= I@2.1.0

### C.168.1. Encoding




### C.168.2. Description

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

### C.168.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.168.4. Decode Variables

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

### C.168.5. IDL Operation

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

### C.168.6. Sail Operation

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

### C.168.7. Exceptions

This instruction does not generate synchronous exceptions.

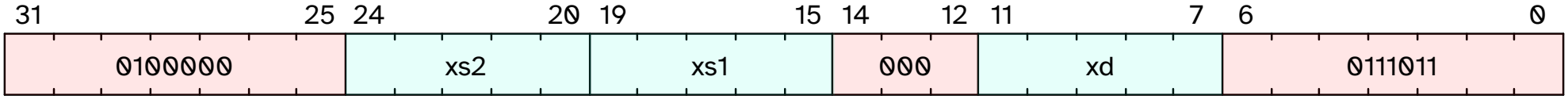
## C.169. subw

**Subtract word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.169.1. Encoding




### C.169.2. Description

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

### C.169.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.169.4. Decode Variables

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

### C.169.5. IDL Operation

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

### C.169.6. Sail Operation

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

### C.169.7. Exceptions

This instruction does not generate synchronous exceptions.

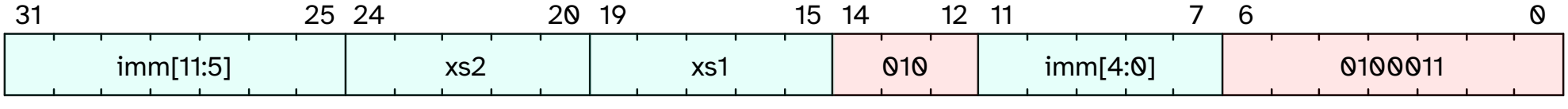
## C.170. sw

**Store word**

This instruction is defined by:

- I, version >= I@2.1.0

### C.170.1. Encoding




### C.170.2. Description

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

### C.170.3. Access

| M | S | U |
|---|---|---|
| Always | Always | Always |

### C.170.4. Decode Variables

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

### C.170.5. IDL Operation

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

### C.170.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
  /* Get the address, X(xs1) + offset.
     Some extensions perform additional checks on address validity. */
  match ext_data_get_addr(xs1, offset, Write(Data), width) {
    Ext_DataAddr_Error(e)  => { ext_handle_data_check_error(e); RETIRE_FAIL },
    Ext_DataAddr_OK(vaddr) =>
      if   check_misaligned(vaddr, width)
  then { handle_mem_exception(vaddr, E_SAMO_Addr_Align()); RETIRE_FAIL }
  else match translateAddr(vaddr, Write(Data)) {
    TR_Failure(e, _)    => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
    TR_Address(paddr, _) => {
      let eares : MemoryOpResult(unit) = match width {
        BYTE   => mem_write_ea(paddr, 1, aq, rl, false),
        HALF   => mem_write_ea(paddr, 2, aq, rl, false),
        WORD   => mem_write_ea(paddr, 4, aq, rl, false),
        DOUBLE => mem_write_ea(paddr, 8, aq, rl, false)
      };
      match (eares) {
        MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL },
        MemValue(_) => {
          let xs2_val = X(xs2);
          let res : MemoryOpResult(bool) = match (width) {
            BYTE => mem_write_value(paddr, 1, xs2_val[7..0],  aq, rl, false),
            HALF => mem_write_value(paddr, 2, xs2_val[15..0], aq, rl, false),
            WORD => mem_write_value(paddr, 4, xs2_val[31..0], aq, rl, false),
            DOUBLE if sizeof(xlen) >= 64
                 => mem_write_value(paddr, 8, xs2_val,        aq, rl, false),
            _    => report_invalid_width(__FILE__, __LINE__, width, "store"),
          };
          match (res) {
```

```
                MemValue(true)  => RETIRE_SUCCESS,
                MemValue(false) => internal_error(__FILE__, __LINE__, "store got false from mem_write_value"),
                MemException(e) => { handle_mem_exception(vaddr, e); RETIRE_FAIL }
              }
            }
          }
        }
      }
  }
}
```

### C.170.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

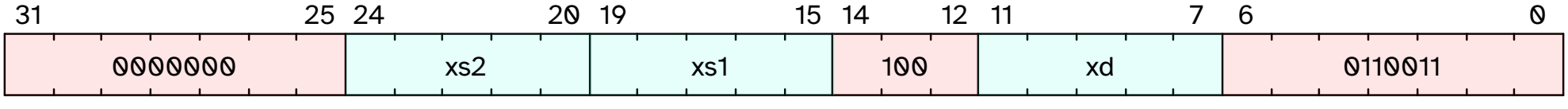
## C.171. xor

**Exclusive Or**

This instruction is defined by:

- I, version >= I@2.1.0

### C.171.1. Encoding




### C.171.2. Description

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

### C.171.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.171.4. Decode Variables

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

### C.171.5. IDL Operation

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

### C.171.6. Sail Operation

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

### C.171.7. Exceptions

This instruction does not generate synchronous exceptions.

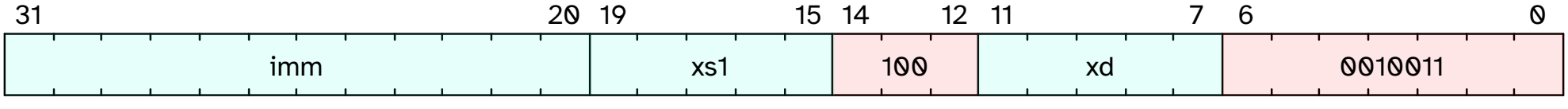
## C.172. xori

### Exclusive Or immediate

This instruction is defined by:

- I, version >= I@2.1.0

### C.172.1. Encoding




### C.172.2. Description

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

### C.172.3. Access

| M | S | U |
| --- | --- | --- |
| Always | Always | Always |

### C.172.4. Decode Variables

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

### C.172.5. IDL Operation

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

### C.172.6. Sail Operation

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

### C.172.7. Exceptions

This instruction does not generate synchronous exceptions.

# Appendix D: CSR Details

## D.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 |

### D.1.1. Attributes

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

### D.1.2. Format

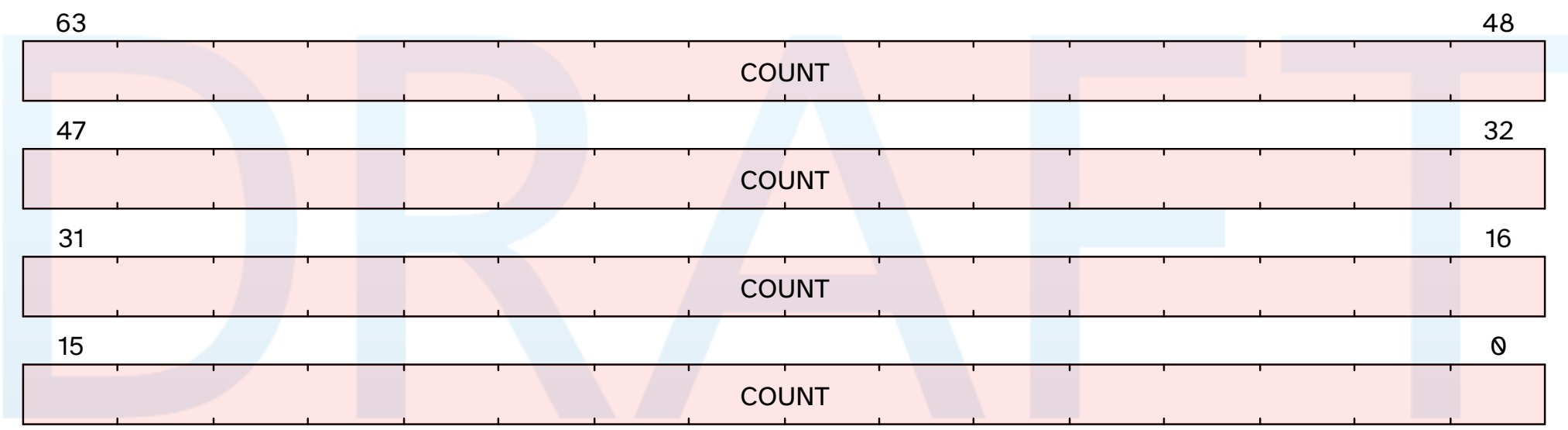



*Figure 1. cycle format*

### D.1.3. Field Summary

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

### D.1.4. Fields

#### cycle.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mcycle.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.1.5. Software read

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

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

## D.2. cycleh

### High-half cycle counter for RDCYCLE Instruction

*cycleh is only defined in RV32.*

Alias for M-mode CSR mcycleh.

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

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

### D.2.1. Attributes

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

### D.2.2. Format

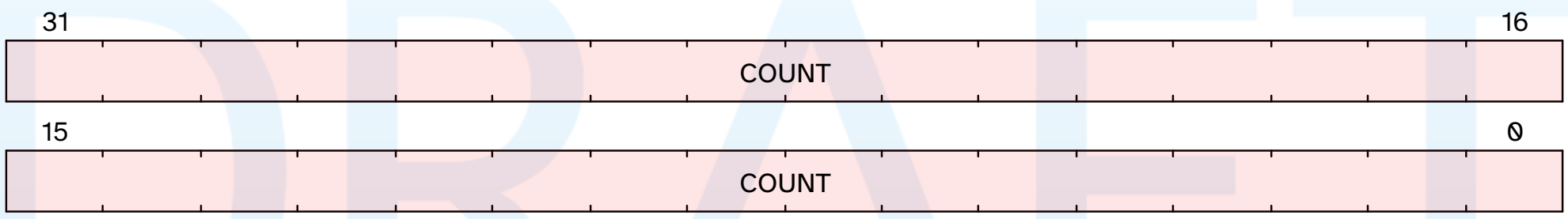



*Figure 2. cycleh format*

### D.2.3. Field Summary

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

### D.2.4. Fields

**cycleh.COUNT Field**

**Location:**

*31:0*

**Description:**

*Alias of mcycleh.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.2.5. Software read

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

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

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

## D.3. fcsr

**Floating-point control and status register (frm + fflags)**

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

*Floating-point control and status register*

Unresolved directive in RVI20ProfileRelease.adoc - include::images/wavedrom/float-csr.adoc[]

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

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

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

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

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

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

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

*Table 13. Accrued exception flag encoding.*

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

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

### D.3.1. Attributes

| | |
|---|---|
| **CSR Address** | 0x3 |
| **Defining extension** | • F, version >= F@2.2.0 |
| **Length** | 32-bit |
| **Privilege Mode** | U |

### D.3.2. Format

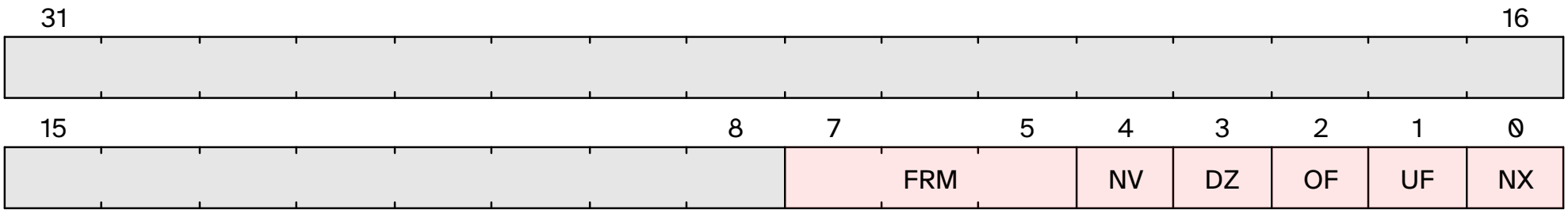



*Figure 3. fcsr format*

### D.3.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| fcsr.FRM | 7:5 | RW-H | UNDEFINED_LEGAL |
| fcsr.NV | 4 | RW-H | UNDEFINED_LEGAL |
| fcsr.DZ | 3 | RW-H | UNDEFINED_LEGAL |
| fcsr.OF | 2 | RW-H | UNDEFINED_LEGAL |
| fcsr.UF | 1 | RW-H | UNDEFINED_LEGAL |
| fcsr.NX | 0 | RW-H | UNDEFINED_LEGAL |

### D.3.4. Fields

#### fcsr.FRM Field

**Location:**

*7:5*

**Description:**

*Rounding modes are encoded as follows:*

*.Rounding mode encoding.*
*[%autowidth,float="center",align="center",cols=",<",options="header"]*
*!===*
*!Rounding Mode |Mnemonic |Meaning*
*!000 !RNE !Round to Nearest, ties to Even*
*!001 !RTZ !Round towards Zero*
*!010 !RDN !Round Down (towards `-\infty`)*
*!011 !RUP !Round Up (towards `+\infty`)*
*!100 !RMM !Round to Nearest, ties to Max Magnitude*
*!101 ! !Reserved for future use.*
*!110 ! !Reserved for future use.*
*!111 !DYN !In instruction's rm field, selects dynamic rounding mode; In Rounding Mode register, reserved.*
*!===*

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

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### fcsr.NV Field

**Location:**

*4*

**Description:**

**Invalid Operation**

*Cumulative error flag for floating point operations.*

*Set by hardware when a floating point operation is invalid and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### fcsr.DZ Field

**Location:**

*3*

**Description:**

**Divide by zero**

*Cumulative error flag for floating point operations.*

*Set by hardware when a floating point divide attempts to divide by zero and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### fcsr.OF Field

**Location:**

*2*

**Description:**

**Overflow**

*Cumulative error flag for floating point operations.*

*Set by hardware when a floating point operation overflows and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### fcsr.UF Field

**Location:**

*1*

**Description:**

**Underflow**

*Cumulative error flag for floating point operations.*

*Set by hardware when a floating point operation underflows and stays set until explicitly*

*cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

#### fcsr.NX Field

**Location:**

0

**Description:**

**Inexact**

*Cumulative error flag for floating point operations.*

*Set by hardware when a floating point operation is inexact and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

## D.4. fflags

### Floating-Point Accrued Exceptions

The accrued exception flags indicate the exception conditions that have arisen on any floating-point arithmetic instruction since the field was last reset by software.

The base RISC-V ISA does not support generating a trap on the setting of a floating-point exception flag.

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

### D.4.1. Attributes

| | |
|---|---|
| **CSR Address** | 0x1 |
| **Defining extension** | • F, version >= F@2.2.0 |
| **Length** | 32-bit |
| **Privilege Mode** | U |

### D.4.2. Format


*Figure 4. fflags format*

### D.4.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| fflags.NV | 4 | RW-H | UNDEFINED_LEGAL |
| fflags.DZ | 3 | RW-H | UNDEFINED_LEGAL |
| fflags.OF | 2 | RW-H | UNDEFINED_LEGAL |
| fflags.UF | 1 | RW-H | UNDEFINED_LEGAL |
| fflags.NX | 0 | RW-H | UNDEFINED_LEGAL |

### D.4.4. Fields

#### fflags.NV Field

**Location:**

*4*

**Description:**

*Set by hardware when a floating point operation is invalid and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

**fflags.DZ Field**

**Location:**

*3*

**Description:**

*Set by hardware when a floating point divide attempts to divide by zero and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

**fflags.OF Field**

**Location:**

*2*

**Description:**

*Set by hardware when a floating point operation overflows and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

**fflags.UF Field**

**Location:**

*1*

**Description:**

*Set by hardware when a floating point operation underflows and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

**fflags.NX Field**

**Location:**

0

**Description:**

*Set by hardware when a floating point operation is inexact and stays set until explicitly cleared by software.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.4.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
NV = CSR[fcsr].NV = csr_value.NV;
return csr_value.NV;

DZ = CSR[fcsr].DZ = csr_value.DZ;
return csr_value.DZ;

OF = CSR[fcsr].OF = csr_value.OF;
return csr_value.OF;

UF = CSR[fcsr].UF = csr_value.UF;
return csr_value.UF;

NX = CSR[fcsr].NX = csr_value.NX;
return csr_value.NX;
```

### D.4.6. Software read

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

```
return (fcsr.NV `<< 4) | (fcsr.DZ `<< 3) | (fcsr.OF `<< 2) | (fcsr.UF `<< 1) | fcsr.NX;
```

## D.5. frm

### Floating-Point Dynamic Rounding Mode

Rounding modes are encoded as follows:

*Table 14. Rounding mode encoding.*

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

The behavior of floating-point instructions that depend on rounding mode when executed with a reserved rounding mode is *reserved*, including both static reserved rounding modes (101-110) and dynamic reserved rounding modes (101-111).

Some instructions, including widening conversions, have the *rm* field but are nevertheless mathematically unaffected by the rounding mode; software should set their *rm* field to RNE (000) but implementations must treat the *rm* field as usual (in particular, with regard to decoding legal vs. reserved encodings).

### D.5.1. Attributes

| | |
|---|---|
| **CSR Address** | 0x2 |
| **Defining extension** | • F, version >= F@2.2.0 |
| **Length** | 32-bit |
| **Privilege Mode** | U |

### D.5.2. Format

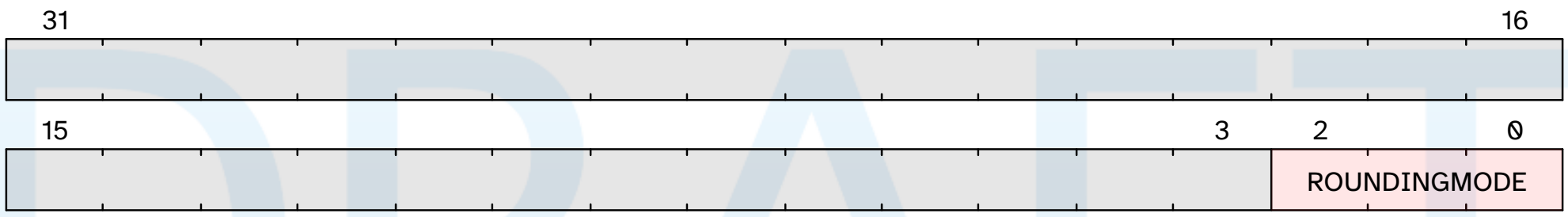



*Figure 5. frm format*

### D.5.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| frm.ROUNDINGMODE | 2:0 | RW-H | UNDEFINED_LEGAL |

### D.5.4. Fields

**frm.ROUNDINGMODE Field**

**Location:**

*2:0*

**Description:**

*Rounding mode data.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.5.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
ROUNDINGMODE = CSR[fcsr].FRM = csr_value.ROUNDINGMODE;
return csr_value.ROUNDINGMODE;
```

### D.5.6. Software read

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

```
return fcsr.FRM;
```

## D.6. hpmcounter10

**User-mode Hardware Performance Counter 7**

Alias for M-mode CSR mhpmcounter10.

Privilege mode access is controlled with mcounteren.HPM10 <%- if ext?(:S) -%> , scounteren.HPM10 <%- if ext?(:H) -%> , and hcounteren.HPM10 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM10 | scounteren.HPM10 | hcounteren.HPM10 | hpmcounter10 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | `IllegalInstruction` | `IllegalInstruction` | `IllegalInstruction` | `IllegalInstruction` |
| 1 | 0 | 0 | read-only | `IllegalInstruction` | `VirtualInstruction` | `VirtualInstruction` |
| 1 | 1 | 0 | read-only | read-only | `VirtualInstruction` | `VirtualInstruction` |
| 1 | 0 | 1 | read-only | `IllegalInstruction` | read-only | `VirtualInstruction` |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM10 | scounteren.HPM10 | hpmcounter10 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | `IllegalInstruction` | `IllegalInstruction` |
| 1 | 0 | read-only | `IllegalInstruction` |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM10 | hpmcounter10 behavior |
|---|---|
| | U-mode |
| 0 | `IllegalInstruction` |
| 1 | read-only |

<%- end -%>

### D.6.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc0a |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.6.2. Format

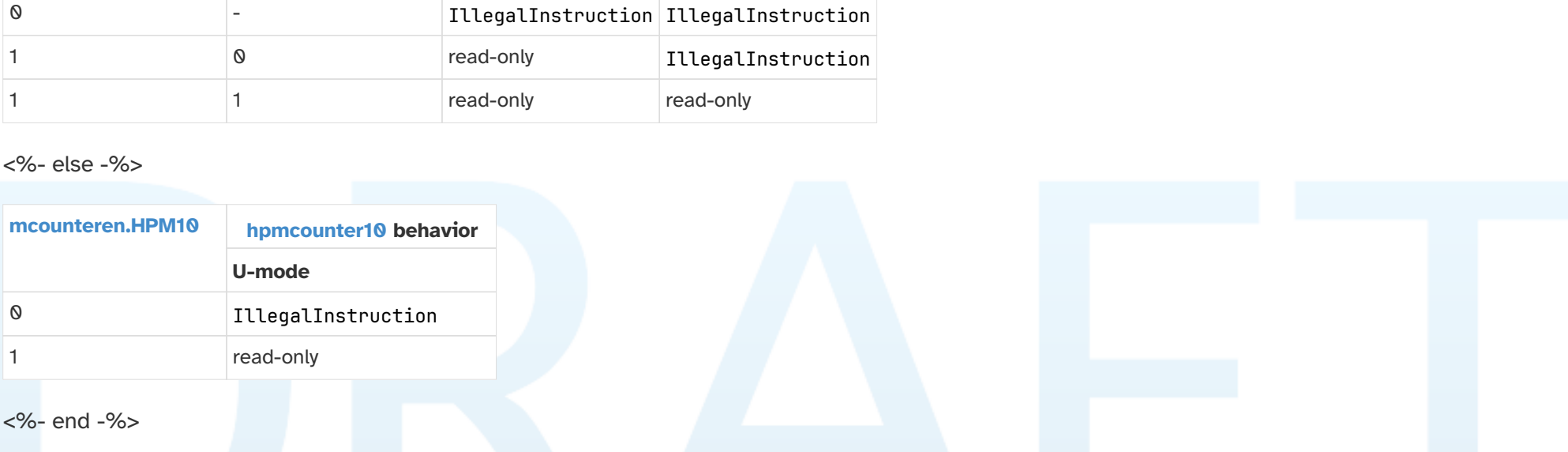



*Figure 6. hpmcounter10 format*

### D.6.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter10.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.6.4. Fields

#### hpmcounter10.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter10.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.6.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM10 & scounteren.HPM10) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM10 == 1'b0 && mcounteren.HPM10 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM10 & scounteren.HPM10) == 1'b0) && (mcounteren.HPM10 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(10);
```

## D.7. hpmcounter11

### User-mode Hardware Performance Counter 8

Alias for M-mode CSR mhpmcounter11.

Privilege mode access is controlled with mcounteren.HPM11 <%- if ext?(:S) -%> , scounteren.HPM11 <%- if ext?(:H) -%> , and hcounteren.HPM11 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM11 | scounteren.HPM11 | hcounteren.HPM11 | hpmcounter11 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | `IllegalInstruction` | `IllegalInstruction` | `IllegalInstruction` | `IllegalInstruction` |
| 1 | 0 | 0 | read-only | `IllegalInstruction` | `VirtualInstruction` | `VirtualInstruction` |
| 1 | 1 | 0 | read-only | read-only | `VirtualInstruction` | `VirtualInstruction` |
| 1 | 0 | 1 | read-only | `IllegalInstruction` | read-only | `VirtualInstruction` |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM11 | scounteren.HPM11 | hpmcounter11 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | `IllegalInstruction` | `IllegalInstruction` |
| 1 | 0 | read-only | `IllegalInstruction` |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM11 | hpmcounter11 behavior |
|---|---|
| | U-mode |
| 0 | `IllegalInstruction` |
| 1 | read-only |

<%- end -%>

### D.7.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc0b |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.7.2. Format

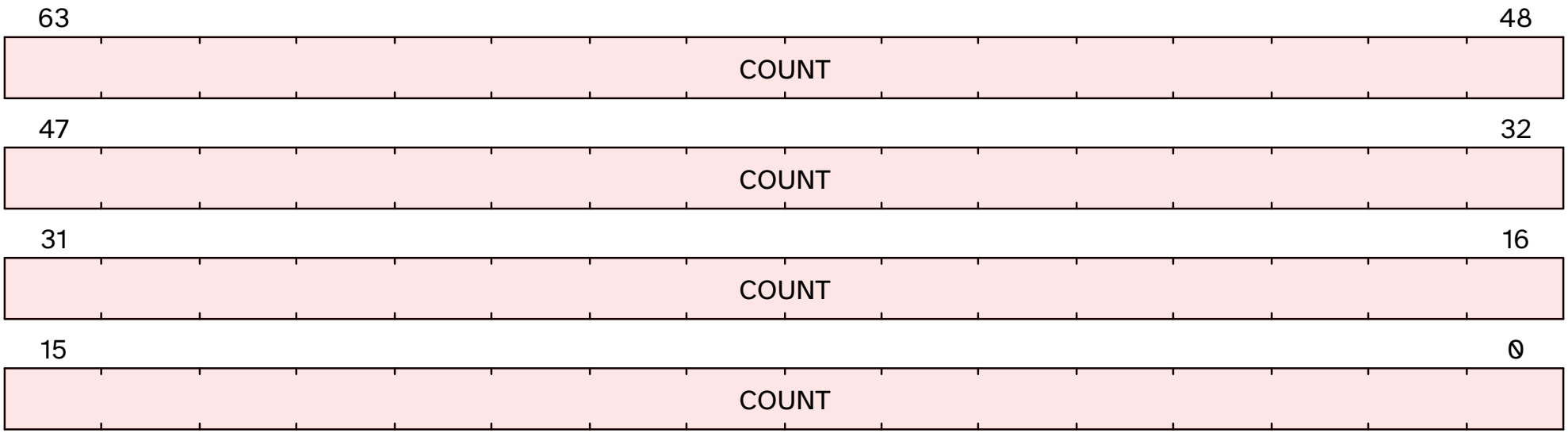



*Figure 7. hpmcounter11 format*

### D.7.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter11.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.7.4. Fields

#### hpmcounter11.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter11.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.7.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM11 & scounteren.HPM11) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM11 == 1'b0 && mcounteren.HPM11 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM11 & scounteren.HPM11) == 1'b0) && (mcounteren.HPM11 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(11);
```

## D.8. hpmcounter12

**User-mode Hardware Performance Counter 9**

Alias for M-mode CSR mhpmcounter12.

Privilege mode access is controlled with mcounteren.HPM12 <%- if ext?(:S) -%> , scounteren.HPM12 <%- if ext?(:H) -%> , and hcounteren.HPM12 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM12 | scounteren.HPM12 | hcounteren.HPM12 | hpmcounter12 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM12 | scounteren.HPM12 | hpmcounter12 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM12 | hpmcounter12 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.8.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc0c |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.8.2. Format

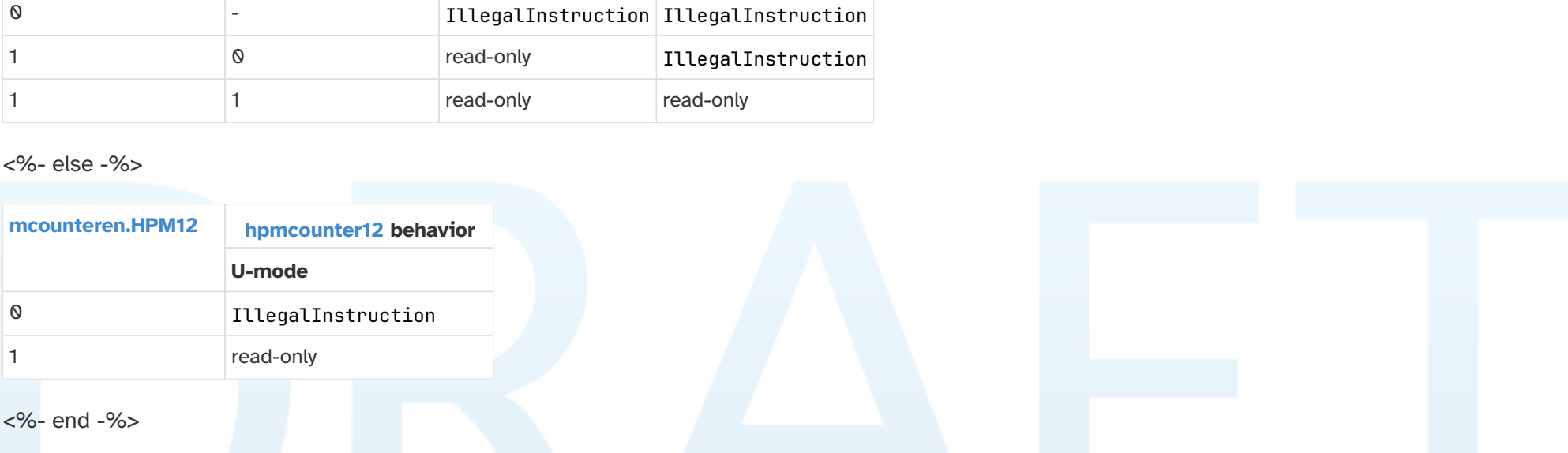



*Figure 8. hpmcounter12 format*

### D.8.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter12.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.8.4. Fields

#### hpmcounter12.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter12.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.8.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM12 & scounteren.HPM12) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM12 == 1'b0 && mcounteren.HPM12 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM12 & scounteren.HPM12) == 1'b0) && (mcounteren.HPM12 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(12);
```

## D.9. hpmcounter13

**User-mode Hardware Performance Counter 10**

Alias for M-mode CSR mhpmcounter13.

Privilege mode access is controlled with mcounteren.HPM13 <%- if ext?(:S) -%> , scounteren.HPM13 <%- if ext?(:H) -%> , and hcounteren.HPM13 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM13 | scounteren.HPM13 | hcounteren.HPM13 | hpmcounter13 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM13 | scounteren.HPM13 | hpmcounter13 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM13 | hpmcounter13 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.9.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc0d |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.9.2. Format

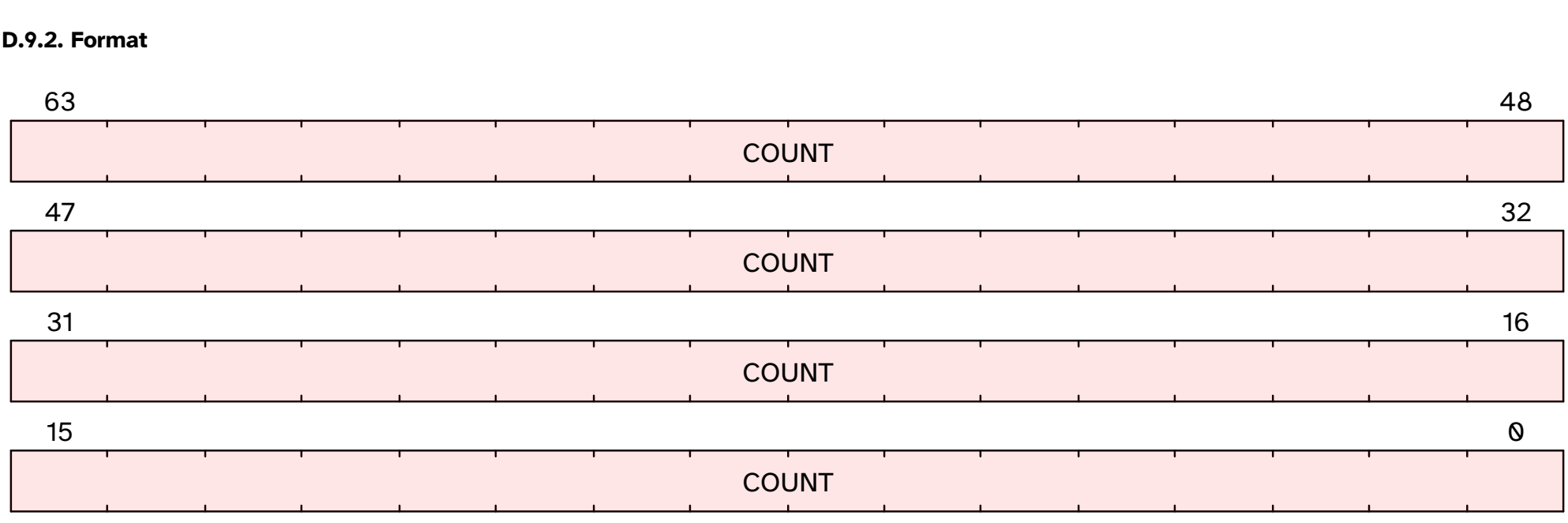



*Figure 9. hpmcounter13 format*

### D.9.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter13.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.9.4. Fields

#### hpmcounter13.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter13.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.9.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM13 & scounteren.HPM13) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM13 == 1'b0 && mcounteren.HPM13 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM13 & scounteren.HPM13) == 1'b0) && (mcounteren.HPM13 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(13);
```

## D.10. hpmcounter14

**User-mode Hardware Performance Counter 11**

Alias for M-mode CSR mhpmcounter14.

Privilege mode access is controlled with mcounteren.HPM14 <%- if ext?(:S) -%> , scounteren.HPM14 <%- if ext?(:H) -%> , and hcounteren.HPM14 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM14 | scounteren.HPM14 | hcounteren.HPM14 | hpmcounter14 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM14 | scounteren.HPM14 | hpmcounter14 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM14 | hpmcounter14 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.10.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc0e |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.10.2. Format

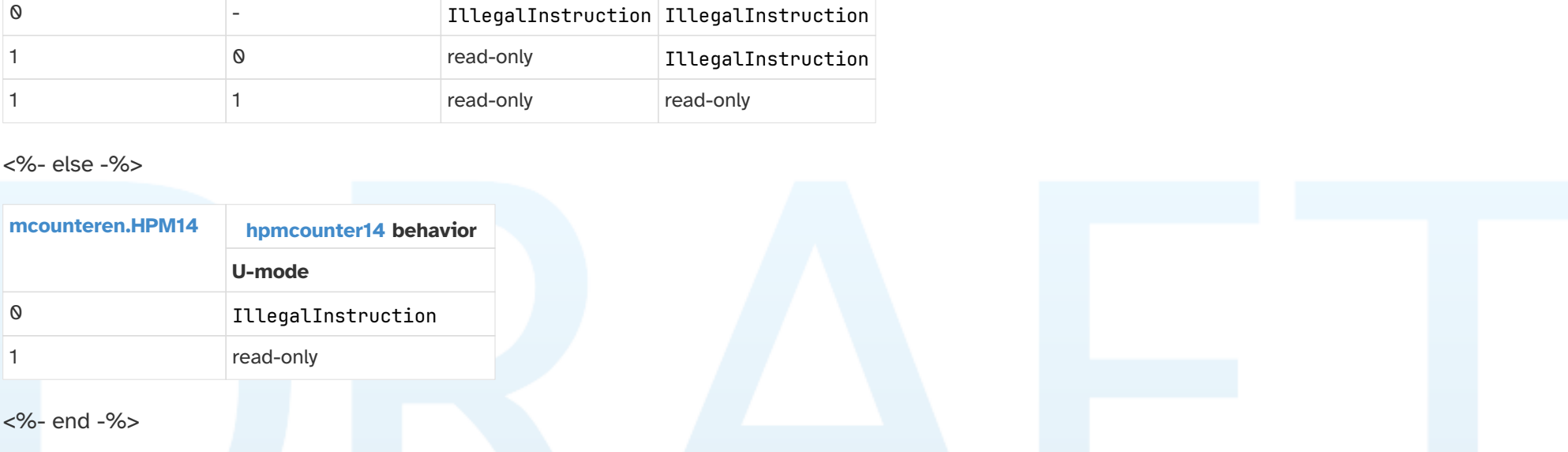



*Figure 10. hpmcounter14 format*

### D.10.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter14.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.10.4. Fields

**hpmcounter14.COUNT Field**

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter14.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.10.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM14 & scounteren.HPM14) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM14 == 1'b0 && mcounteren.HPM14 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM14 & scounteren.HPM14) == 1'b0) && (mcounteren.HPM14 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(14);
```

## D.11. hpmcounter15

**User-mode Hardware Performance Counter 12**

Alias for M-mode CSR mhpmcounter15.

Privilege mode access is controlled with mcounteren.HPM15 <%- if ext?(:S) -%> , scounteren.HPM15 <%- if ext?(:H) -%> , and hcounteren.HPM15 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM15 | scounteren.HPM15 | hcounteren.HPM15 | hpmcounter15 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM15 | scounteren.HPM15 | hpmcounter15 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM15 | hpmcounter15 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.11.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc0f |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.11.2. Format


*Figure 11. hpmcounter15 format*

### D.11.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter15.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.11.4. Fields

#### hpmcounter15.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter15.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.11.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM15 & scounteren.HPM15) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM15 == 1'b0 && mcounteren.HPM15 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM15 & scounteren.HPM15) == 1'b0) && (mcounteren.HPM15 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(15);
```

## D.12. hpmcounter16

**User-mode Hardware Performance Counter 13**

Alias for M-mode CSR mhpmcounter16.

Privilege mode access is controlled with mcounteren.HPM16 <%- if ext?(:S) -%> , scounteren.HPM16 <%- if ext?(:H) -%> , and hcounteren.HPM16 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM16 | scounteren.HPM16 | hcounteren.HPM16 | hpmcounter16 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM16 | scounteren.HPM16 | hpmcounter16 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM16 | hpmcounter16 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.12.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc10 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.12.2. Format

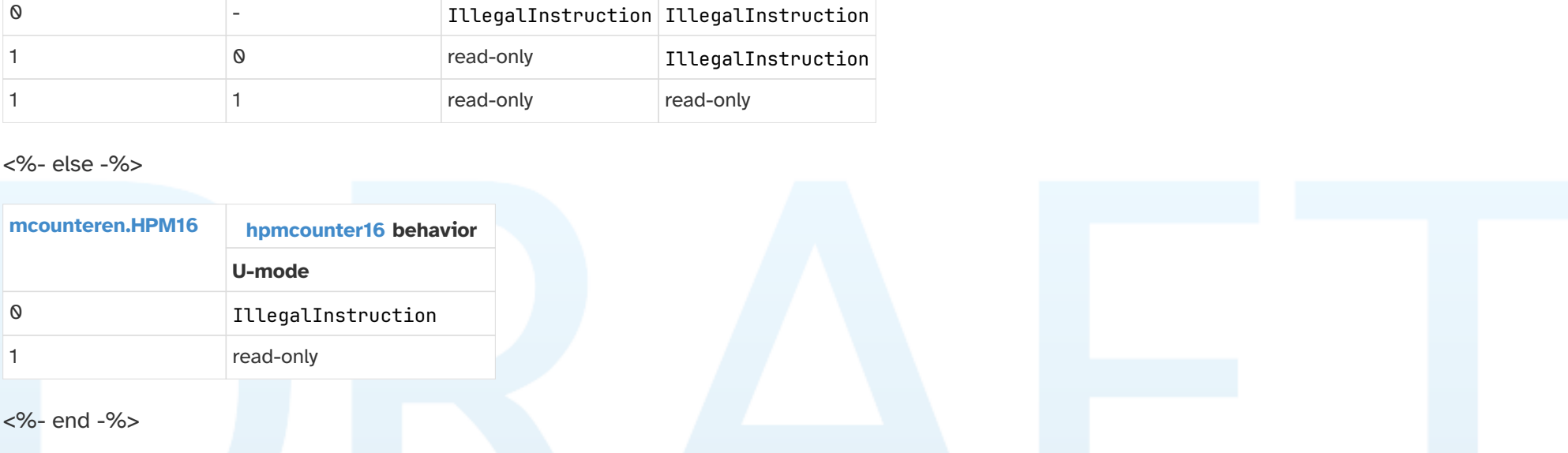



*Figure 12. hpmcounter16 format*

### D.12.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter16.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.12.4. Fields

#### hpmcounter16.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter16.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.12.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM16 & scounteren.HPM16) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM16 == 1'b0 && mcounteren.HPM16 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM16 & scounteren.HPM16) == 1'b0) && (mcounteren.HPM16 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(16);
```

## D.13. hpmcounter17

**User-mode Hardware Performance Counter 14**

Alias for M-mode CSR mhpmcounter17.

Privilege mode access is controlled with mcounteren.HPM17 <%- if ext?(:S) -%> , scounteren.HPM17 <%- if ext?(:H) -%> , and hcounteren.HPM17 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM17 | scounteren.HPM17 | hcounteren.HPM17 | hpmcounter17 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM17 | scounteren.HPM17 | hpmcounter17 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM17 | hpmcounter17 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.13.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc11 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.13.2. Format

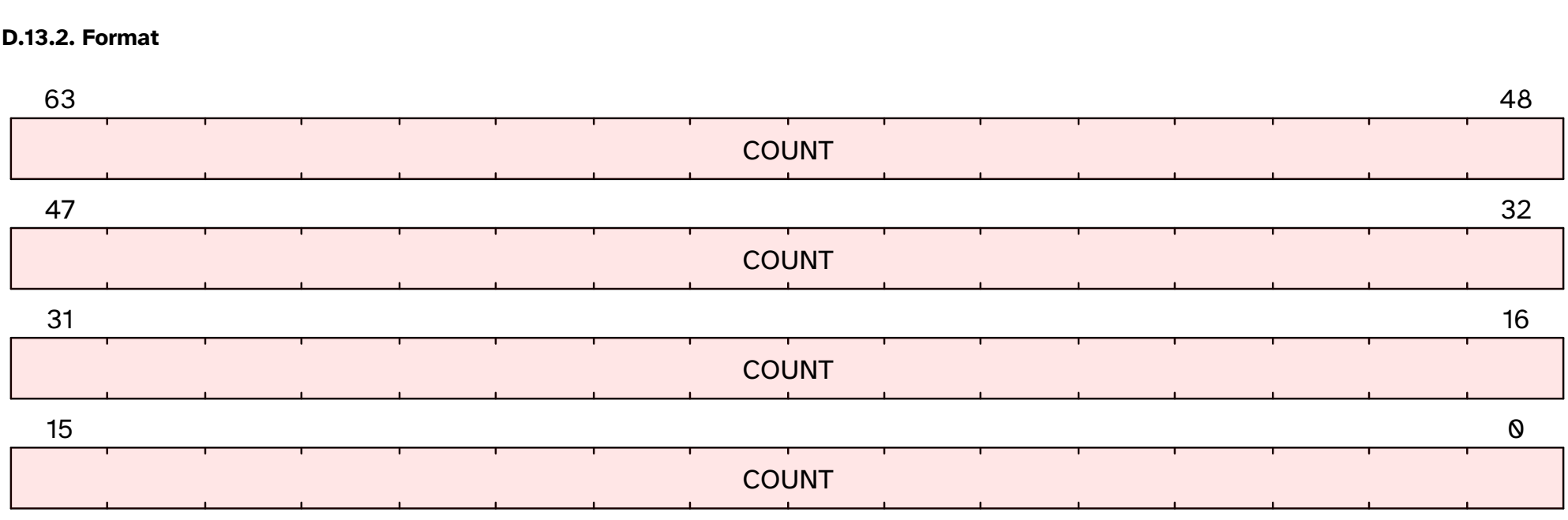



*Figure 13. hpmcounter17 format*

### D.13.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter17.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.13.4. Fields

#### hpmcounter17.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter17.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.13.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM17 & scounteren.HPM17) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM17 == 1'b0 && mcounteren.HPM17 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM17 & scounteren.HPM17) == 1'b0) && (mcounteren.HPM17 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(17);
```

## D.14. hpmcounter18

**User-mode Hardware Performance Counter 15**

Alias for M-mode CSR mhpmcounter18.

Privilege mode access is controlled with mcounteren.HPM18 <%- if ext?(:S) -%> , scounteren.HPM18 <%- if ext?(:H) -%> , and hcounteren.HPM18 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM18 | scounteren.HPM18 | hcounteren.HPM18 | hpmcounter18 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM18 | scounteren.HPM18 | hpmcounter18 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM18 | hpmcounter18 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.14.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc12 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.14.2. Format


*Figure 14. hpmcounter18 format*

### D.14.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter18.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.14.4. Fields

#### hpmcounter18.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter18.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.14.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM18 & scounteren.HPM18) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM18 == 1'b0 && mcounteren.HPM18 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM18 & scounteren.HPM18) == 1'b0) && (mcounteren.HPM18 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(18);
```

## D.15. hpmcounter19

**User-mode Hardware Performance Counter 16**

Alias for M-mode CSR mhpmcounter19.

Privilege mode access is controlled with mcounteren.HPM19 <%- if ext?(:S) -%> , scounteren.HPM19 <%- if ext?(:H) -%> , and hcounteren.HPM19 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM19 | scounteren.HPM19 | hcounteren.HPM19 | hpmcounter19 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM19 | scounteren.HPM19 | hpmcounter19 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM19 | hpmcounter19 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.15.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc13 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.15.2. Format

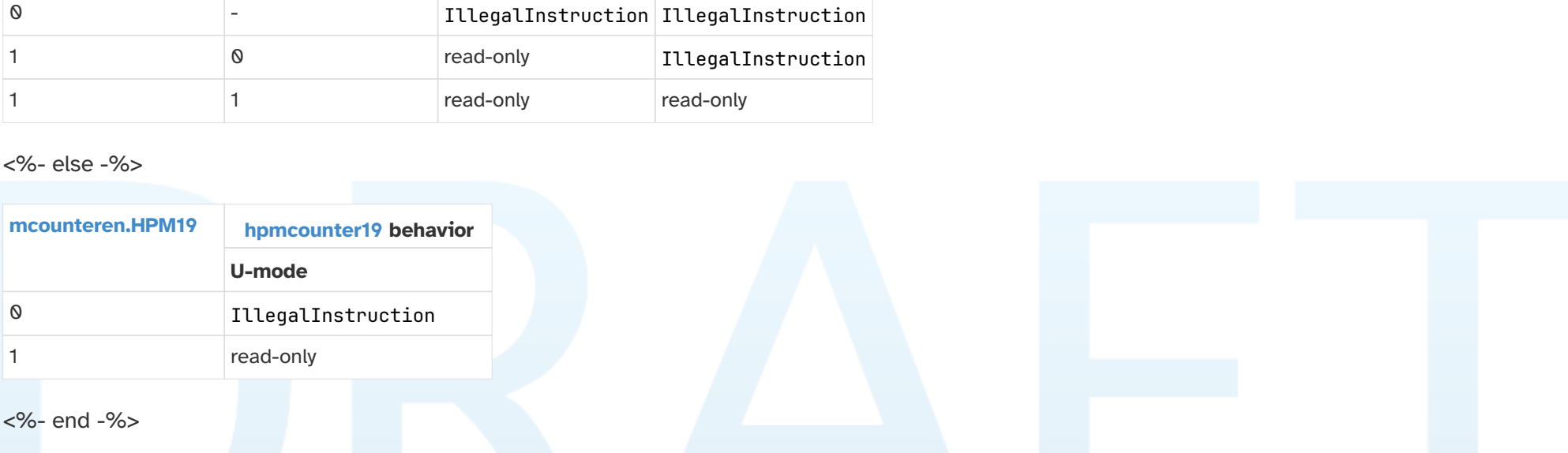



*Figure 15. hpmcounter19 format*

### D.15.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter19.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.15.4. Fields

#### hpmcounter19.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter19.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.15.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM19 & scounteren.HPM19) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM19 == 1'b0 && mcounteren.HPM19 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM19 & scounteren.HPM19) == 1'b0) && (mcounteren.HPM19 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(19);
```

## D.16. hpmcounter20

**User-mode Hardware Performance Counter 17**

Alias for M-mode CSR mhpmcounter20.

Privilege mode access is controlled with mcounteren.HPM20 <%- if ext?(:S) -%> , scounteren.HPM20 <%- if ext?(:H) -%> , and hcounteren.HPM20 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM20 | scounteren.HPM20 | hcounteren.HPM20 | hpmcounter20 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM20 | scounteren.HPM20 | hpmcounter20 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM20 | hpmcounter20 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.16.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc14 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.16.2. Format

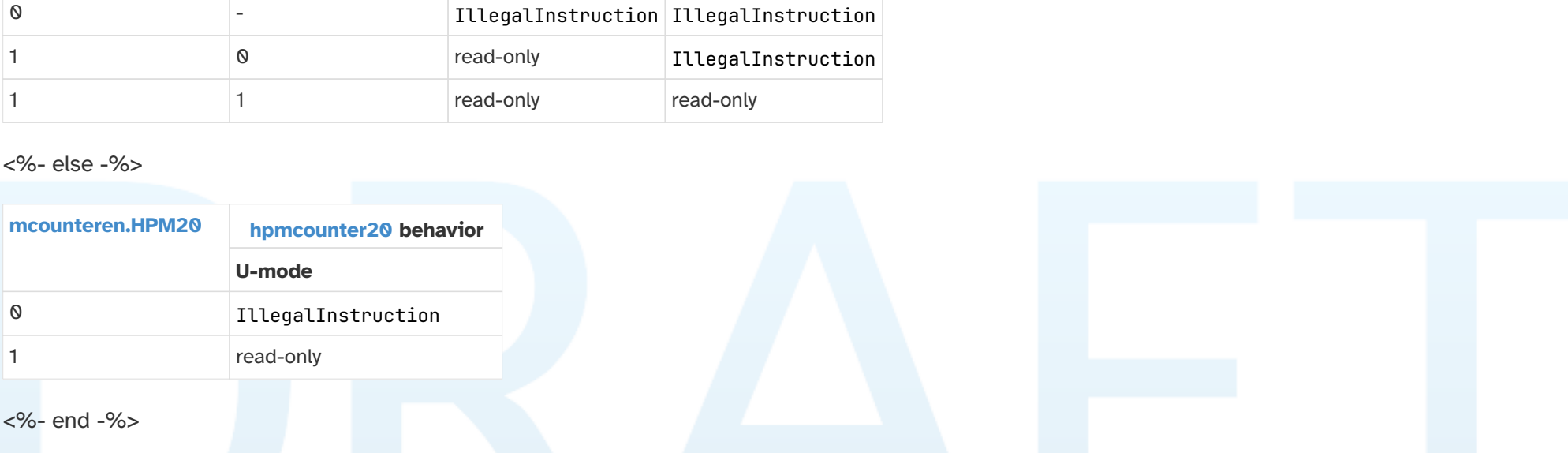



*Figure 16. hpmcounter20 format*

### D.16.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter20.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.16.4. Fields

#### hpmcounter20.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter20.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.16.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM20 & scounteren.HPM20) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM20 == 1'b0 && mcounteren.HPM20 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM20 & scounteren.HPM20) == 1'b0) && (mcounteren.HPM20 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(20);
```

## D.17. hpmcounter21

**User-mode Hardware Performance Counter 18**

Alias for M-mode CSR mhpmcounter21.

Privilege mode access is controlled with mcounteren.HPM21 <%- if ext?(:S) -%> , scounteren.HPM21 <%- if ext?(:H) -%> , and hcounteren.HPM21 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM21 | scounteren.HPM21 | hcounteren.HPM21 | hpmcounter21 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM21 | scounteren.HPM21 | hpmcounter21 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM21 | hpmcounter21 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.17.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc15 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.17.2. Format

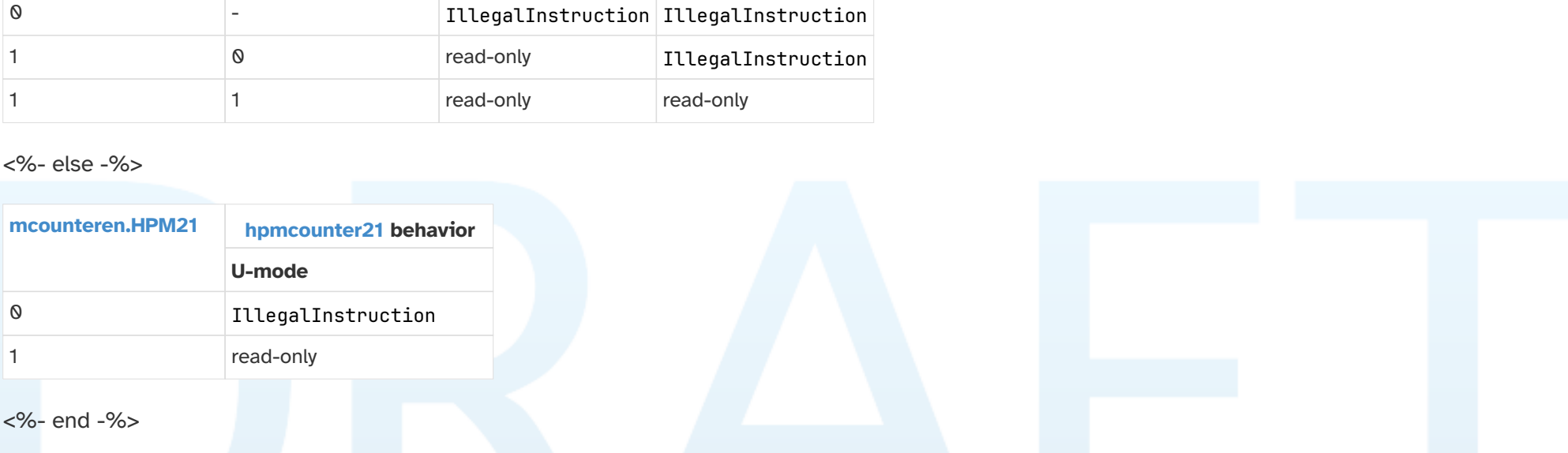



*Figure 17. hpmcounter21 format*

### D.17.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter21.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.17.4. Fields

#### hpmcounter21.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter21.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.17.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM21 & scounteren.HPM21) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM21 == 1'b0 && mcounteren.HPM21 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM21 & scounteren.HPM21) == 1'b0) && (mcounteren.HPM21 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(21);
```

## D.18. hpmcounter22

**User-mode Hardware Performance Counter 19**

Alias for M-mode CSR mhpmcounter22.

Privilege mode access is controlled with mcounteren.HPM22 <%- if ext?(:S) -%> , scounteren.HPM22 <%- if ext?(:H) -%> , and hcounteren.HPM22 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM22 | scounteren.HPM22 | hcounteren.HPM22 | hpmcounter22 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM22 | scounteren.HPM22 | hpmcounter22 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM22 | hpmcounter22 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.18.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc16 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.18.2. Format


*Figure 18. hpmcounter22 format*

### D.18.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter22.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.18.4. Fields

#### hpmcounter22.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter22.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.18.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM22 & scounteren.HPM22) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM22 == 1'b0 && mcounteren.HPM22 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM22 & scounteren.HPM22) == 1'b0) && (mcounteren.HPM22 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(22);
```

## D.19. hpmcounter23

**User-mode Hardware Performance Counter 20**

Alias for M-mode CSR mhpmcounter23.

Privilege mode access is controlled with mcounteren.HPM23 <%- if ext?(:S) -%> , scounteren.HPM23 <%- if ext?(:H) -%> , and hcounteren.HPM23 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM23 | scounteren.HPM23 | hcounteren.HPM23 | hpmcounter23 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM23 | scounteren.HPM23 | hpmcounter23 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM23 | hpmcounter23 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.19.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc17 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.19.2. Format

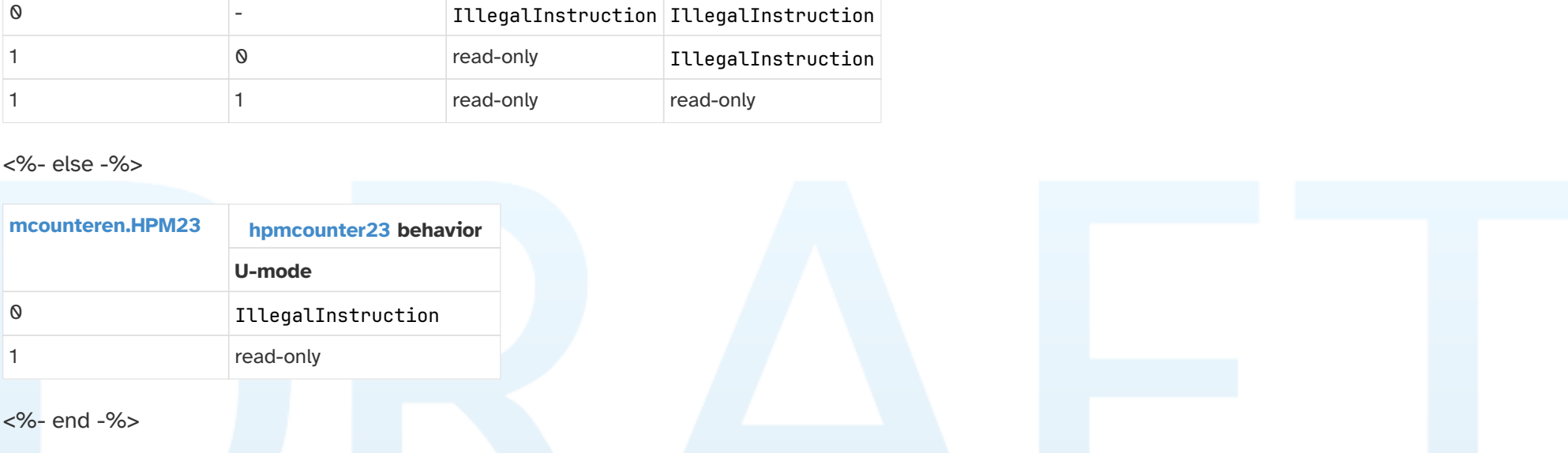



*Figure 19. hpmcounter23 format*

### D.19.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter23.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.19.4. Fields

#### hpmcounter23.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter23.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.19.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM23 & scounteren.HPM23) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM23 == 1'b0 && mcounteren.HPM23 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM23 & scounteren.HPM23) == 1'b0) && (mcounteren.HPM23 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(23);
```

## D.20. hpmcounter24

**User-mode Hardware Performance Counter 21**

Alias for M-mode CSR mhpmcounter24.

Privilege mode access is controlled with mcounteren.HPM24 <%- if ext?(:S) -%> , scounteren.HPM24 <%- if ext?(:H) -%> , and hcounteren.HPM24 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM24 | scounteren.HPM24 | hcounteren.HPM24 | hpmcounter24 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM24 | scounteren.HPM24 | hpmcounter24 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM24 | hpmcounter24 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.20.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc18 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.20.2. Format

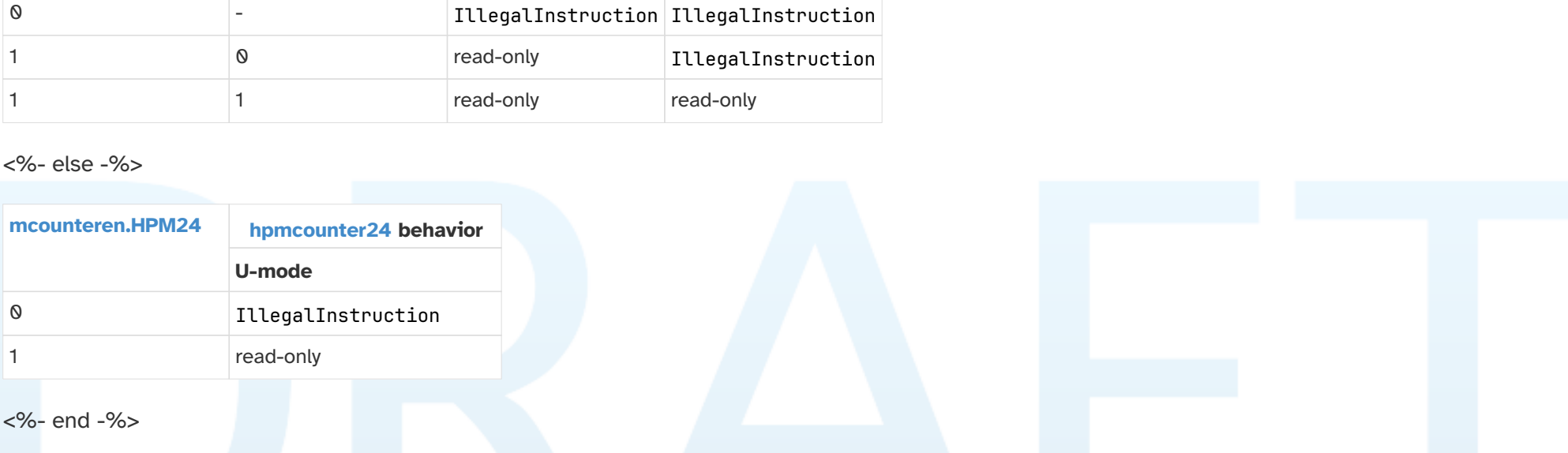



*Figure 20. hpmcounter24 format*

### D.20.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter24.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.20.4. Fields

#### hpmcounter24.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter24.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.20.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM24 & scounteren.HPM24) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM24 == 1'b0 && mcounteren.HPM24 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM24 & scounteren.HPM24) == 1'b0) && (mcounteren.HPM24 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(24);
```

## D.21. hpmcounter25

**User-mode Hardware Performance Counter 22**

Alias for M-mode CSR mhpmcounter25.

Privilege mode access is controlled with mcounteren.HPM25 <%- if ext?(:S) -%> , scounteren.HPM25 <%- if ext?(:H) -%> , and hcounteren.HPM25 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM25 | scounteren.HPM25 | hcounteren.HPM25 | hpmcounter25 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM25 | scounteren.HPM25 | hpmcounter25 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM25 | hpmcounter25 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.21.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc19 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.21.2. Format

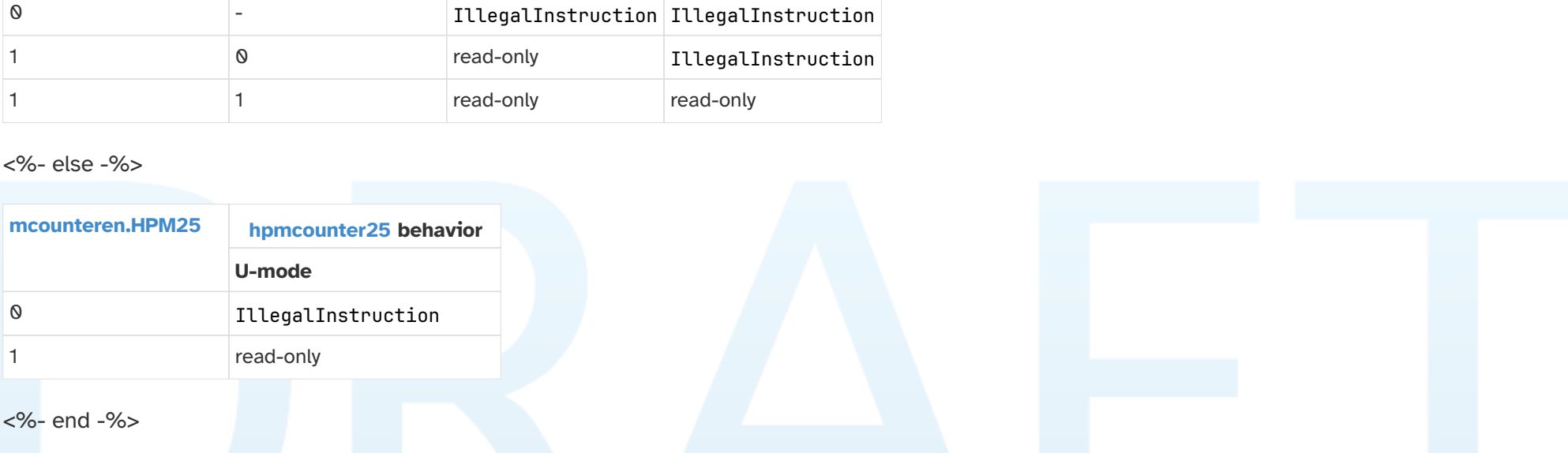



*Figure 21. hpmcounter25 format*

### D.21.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter25.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.21.4. Fields

#### hpmcounter25.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter25.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.21.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM25 & scounteren.HPM25) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM25 == 1'b0 && mcounteren.HPM25 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM25 & scounteren.HPM25) == 1'b0) && (mcounteren.HPM25 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(25);
```

## D.22. hpmcounter26

**User-mode Hardware Performance Counter 23**

Alias for M-mode CSR mhpmcounter26.

Privilege mode access is controlled with mcounteren.HPM26 <%- if ext?(:S) -%> , scounteren.HPM26 <%- if ext?(:H) -%> , and hcounteren.HPM26 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM26 | scounteren.HPM26 | hcounteren.HPM26 | hpmcounter26 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM26 | scounteren.HPM26 | hpmcounter26 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM26 | hpmcounter26 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.22.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc1a |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.22.2. Format


*Figure 22. hpmcounter26 format*

### D.22.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter26.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.22.4. Fields

#### hpmcounter26.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter26.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.22.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM26 & scounteren.HPM26) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM26 == 1'b0 && mcounteren.HPM26 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM26 & scounteren.HPM26) == 1'b0) && (mcounteren.HPM26 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(26);
```

## D.23. hpmcounter27

**User-mode Hardware Performance Counter 24**

Alias for M-mode CSR mhpmcounter27.

Privilege mode access is controlled with mcounteren.HPM27 <%- if ext?(:S) -%> , scounteren.HPM27 <%- if ext?(:H) -%> , and hcounteren.HPM27 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM27 | scounteren.HPM27 | hcounteren.HPM27 | hpmcounter27 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM27 | scounteren.HPM27 | hpmcounter27 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM27 | hpmcounter27 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.23.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc1b |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.23.2. Format

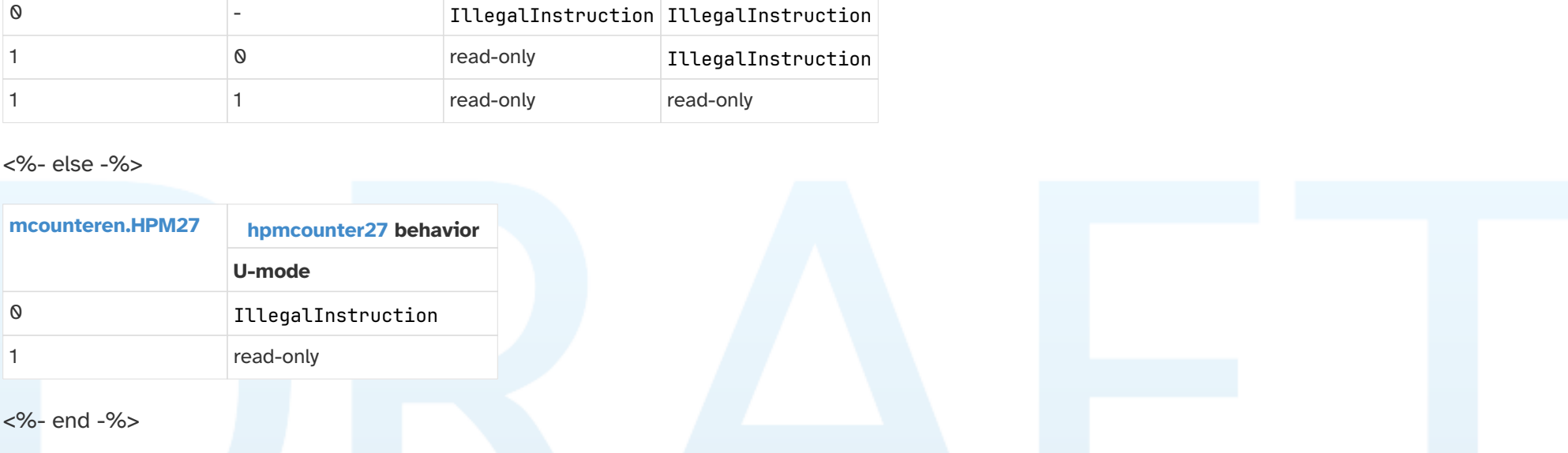



*Figure 23. hpmcounter27 format*

### D.23.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter27.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.23.4. Fields

#### hpmcounter27.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter27.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.23.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM27 & scounteren.HPM27) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM27 == 1'b0 && mcounteren.HPM27 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM27 & scounteren.HPM27) == 1'b0) && (mcounteren.HPM27 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(27);
```

## D.24. hpmcounter28

**User-mode Hardware Performance Counter 25**

Alias for M-mode CSR mhpmcounter28.

Privilege mode access is controlled with mcounteren.HPM28 <%- if ext?(:S) -%> , scounteren.HPM28 <%- if ext?(:H) -%> , and hcounteren.HPM28 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM28 | scounteren.HPM28 | hcounteren.HPM28 | hpmcounter28 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM28 | scounteren.HPM28 | hpmcounter28 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM28 | hpmcounter28 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.24.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc1c |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.24.2. Format

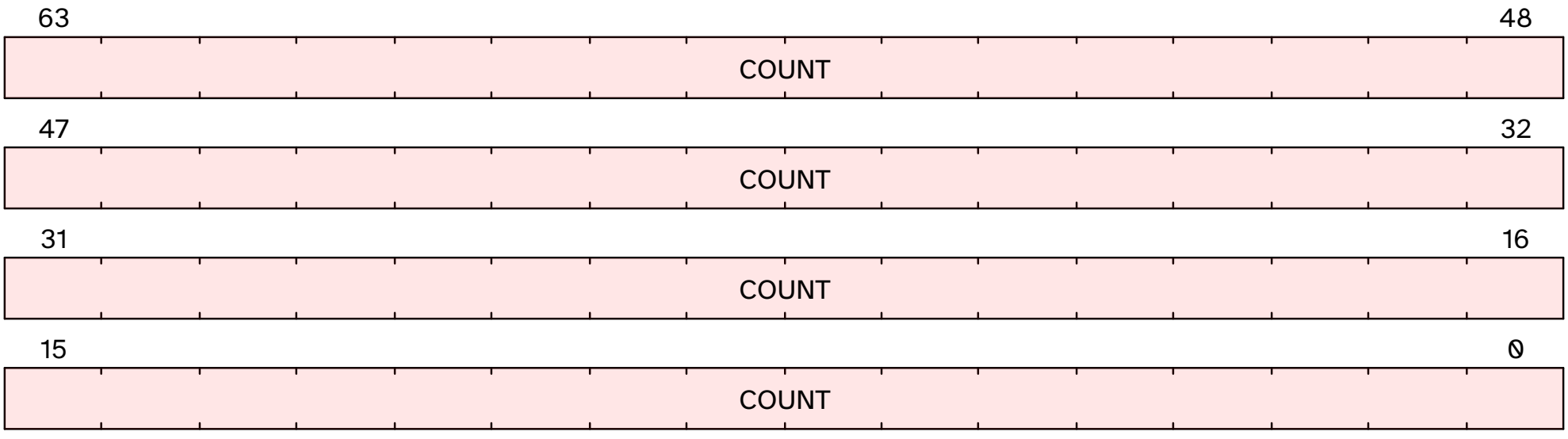



*Figure 24. hpmcounter28 format*

### D.24.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter28.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.24.4. Fields

#### hpmcounter28.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter28.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.24.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM28 & scounteren.HPM28) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM28 == 1'b0 && mcounteren.HPM28 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM28 & scounteren.HPM28) == 1'b0) && (mcounteren.HPM28 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(28);
```

## D.25. hpmcounter29

**User-mode Hardware Performance Counter 26**

Alias for M-mode CSR mhpmcounter29.

Privilege mode access is controlled with mcounteren.HPM29 <%- if ext?(:S) -%> , scounteren.HPM29 <%- if ext?(:H) -%> , and hcounteren.HPM29 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM29 | scounteren.HPM29 | hcounteren.HPM29 | hpmcounter29 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM29 | scounteren.HPM29 | hpmcounter29 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM29 | hpmcounter29 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.25.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc1d |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.25.2. Format

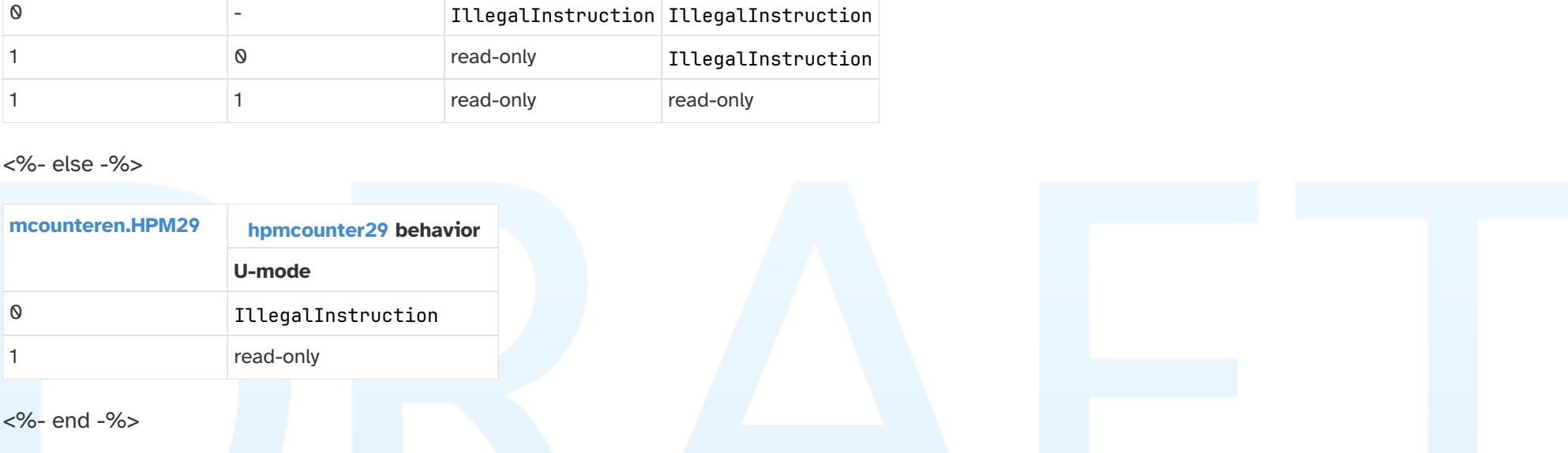



*Figure 25. hpmcounter29 format*

### D.25.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter29.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.25.4. Fields

#### hpmcounter29.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter29.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.25.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM29 & scounteren.HPM29) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM29 == 1'b0 && mcounteren.HPM29 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM29 & scounteren.HPM29) == 1'b0) && (mcounteren.HPM29 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(29);
```

## D.26. hpmcounter3

### User-mode Hardware Performance Counter 0

Alias for M-mode CSR mhpmcounter3.

Privilege mode access is controlled with mcounteren.HPM3 <%- if ext?(:S) -%> , scounteren.HPM3 <%- if ext?(:H) -%> , and hcounteren.HPM3 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM3 | scounteren.HPM3 | hcounteren.HPM3 | hpmcounter3 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM3 | scounteren.HPM3 | hpmcounter3 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM3 | hpmcounter3 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.26.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc03 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.26.2. Format

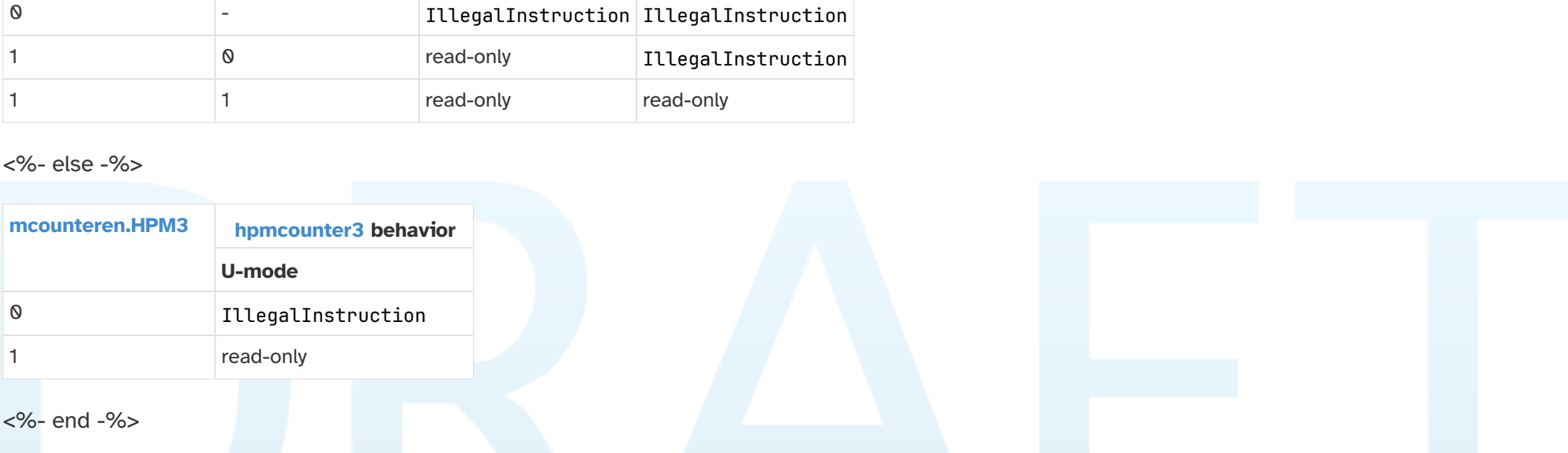



*Figure 26. hpmcounter3 format*

### D.26.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter3.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.26.4. Fields

#### hpmcounter3.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter3.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.26.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM3 & scounteren.HPM3) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM3 == 1'b0 && mcounteren.HPM3 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM3 & scounteren.HPM3) == 1'b0) && (mcounteren.HPM3 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(3);
```

## D.27. hpmcounter30

**User-mode Hardware Performance Counter 27**

Alias for M-mode CSR mhpmcounter30.

Privilege mode access is controlled with mcounteren.HPM30 <%- if ext?(:S) -%> , scounteren.HPM30 <%- if ext?(:H) -%> , and hcounteren.HPM30 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM30 | scounteren.HPM30 | hcounteren.HPM30 | hpmcounter30 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM30 | scounteren.HPM30 | hpmcounter30 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM30 | hpmcounter30 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.27.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc1e |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.27.2. Format


*Figure 27. hpmcounter30 format*

### D.27.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter30.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.27.4. Fields

#### hpmcounter30.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter30.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.27.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM30 & scounteren.HPM30) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM30 == 1'b0 && mcounteren.HPM30 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM30 & scounteren.HPM30) == 1'b0) && (mcounteren.HPM30 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(30);
```

## D.28. hpmcounter31

**User-mode Hardware Performance Counter 28**

Alias for M-mode CSR mhpmcounter31.

Privilege mode access is controlled with mcounteren.HPM31 <%- if ext?(:S) -%> , scounteren.HPM31 <%- if ext?(:H) -%> , and hcounteren.HPM31 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM31 | scounteren.HPM31 | hcounteren.HPM31 | hpmcounter31 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM31 | scounteren.HPM31 | hpmcounter31 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM31 | hpmcounter31 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.28.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc1f |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.28.2. Format


*Figure 28. hpmcounter31 format*

### D.28.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter31.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.28.4. Fields

#### hpmcounter31.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter31.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.28.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM31 & scounteren.HPM31) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM31 == 1'b0 && mcounteren.HPM31 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM31 & scounteren.HPM31) == 1'b0) && (mcounteren.HPM31 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(31);
```

## D.29. hpmcounter4

**User-mode Hardware Performance Counter 1**

Alias for M-mode CSR mhpmcounter4.

Privilege mode access is controlled with mcounteren.HPM4 <%- if ext?(:S) -%> , scounteren.HPM4 <%- if ext?(:H) -%> , and hcounteren.HPM4 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM4 | scounteren.HPM4 | hcounteren.HPM4 | hpmcounter4 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM4 | scounteren.HPM4 | hpmcounter4 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM4 | hpmcounter4 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.29.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc04 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.29.2. Format


*Figure 29. hpmcounter4 format*

### D.29.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter4.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.29.4. Fields

#### hpmcounter4.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter4.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.29.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM4 & scounteren.HPM4) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM4 == 1'b0 && mcounteren.HPM4 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM4 & scounteren.HPM4) == 1'b0) && (mcounteren.HPM4 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(4);
```

## D.30. hpmcounter5

**User-mode Hardware Performance Counter 2**

Alias for M-mode CSR mhpmcounter5.

Privilege mode access is controlled with mcounteren.HPM5 <%- if ext?(:S) -%> , scounteren.HPM5 <%- if ext?(:H) -%> , and hcounteren.HPM5 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM5 | scounteren.HPM5 | hcounteren.HPM5 | hpmcounter5 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM5 | scounteren.HPM5 | hpmcounter5 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM5 | hpmcounter5 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.30.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc05 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.30.2. Format

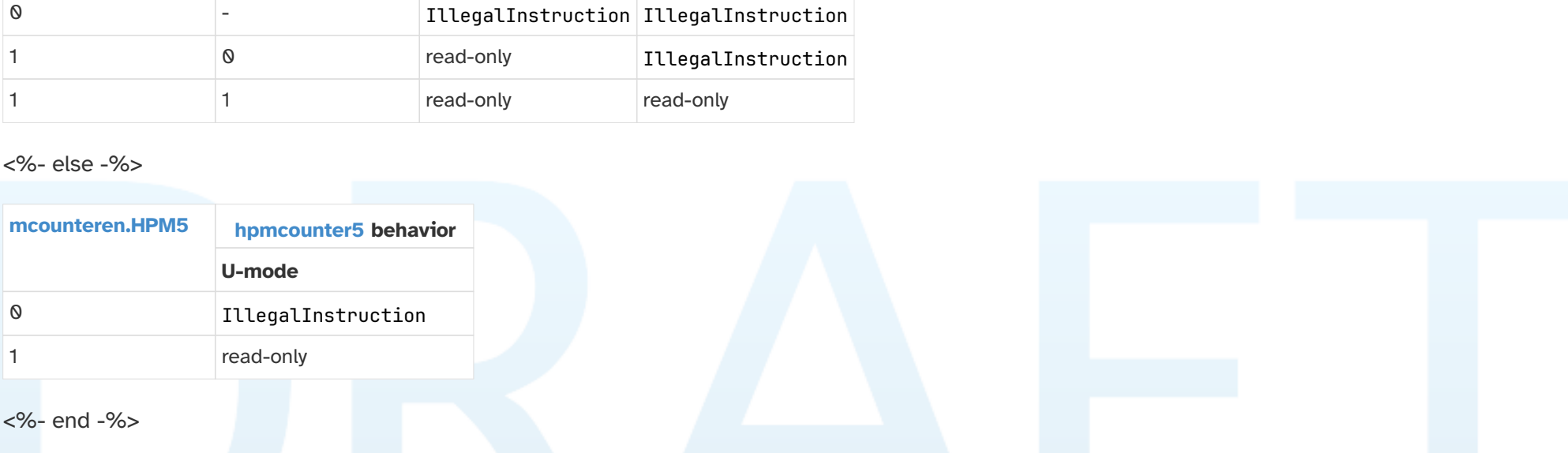



*Figure 30. hpmcounter5 format*

### D.30.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter5.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.30.4. Fields

**hpmcounter5.COUNT Field**

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter5.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.30.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM5 & scounteren.HPM5) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM5 == 1'b0 && mcounteren.HPM5 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM5 & scounteren.HPM5) == 1'b0) && (mcounteren.HPM5 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(5);
```

## D.31. hpmcounter6

**User-mode Hardware Performance Counter 3**

Alias for M-mode CSR mhpmcounter6.

Privilege mode access is controlled with mcounteren.HPM6 <%- if ext?(:S) -%> , scounteren.HPM6 <%- if ext?(:H) -%> , and hcounteren.HPM6 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM6 | scounteren.HPM6 | hcounteren.HPM6 | hpmcounter6 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM6 | scounteren.HPM6 | hpmcounter6 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM6 | hpmcounter6 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.31.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc06 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.31.2. Format

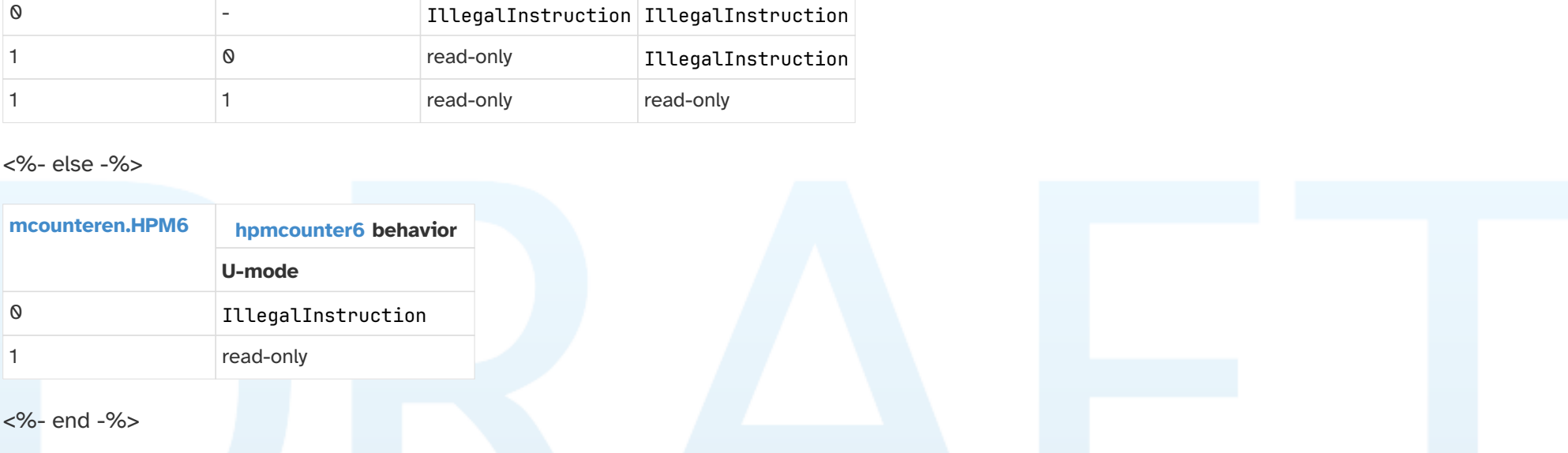



*Figure 31. hpmcounter6 format*

### D.31.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter6.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.31.4. Fields

#### hpmcounter6.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter6.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.31.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM6 & scounteren.HPM6) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM6 == 1'b0 && mcounteren.HPM6 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM6 & scounteren.HPM6) == 1'b0) && (mcounteren.HPM6 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(6);
```

## D.32. hpmcounter7

**User-mode Hardware Performance Counter 4**

Alias for M-mode CSR mhpmcounter7.

Privilege mode access is controlled with mcounteren.HPM7 <%- if ext?(:S) -%> , scounteren.HPM7 <%- if ext?(:H) -%> , and hcounteren.HPM7 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM7 | scounteren.HPM7 | hcounteren.HPM7 | hpmcounter7 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM7 | scounteren.HPM7 | hpmcounter7 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM7 | hpmcounter7 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.32.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc07 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.32.2. Format

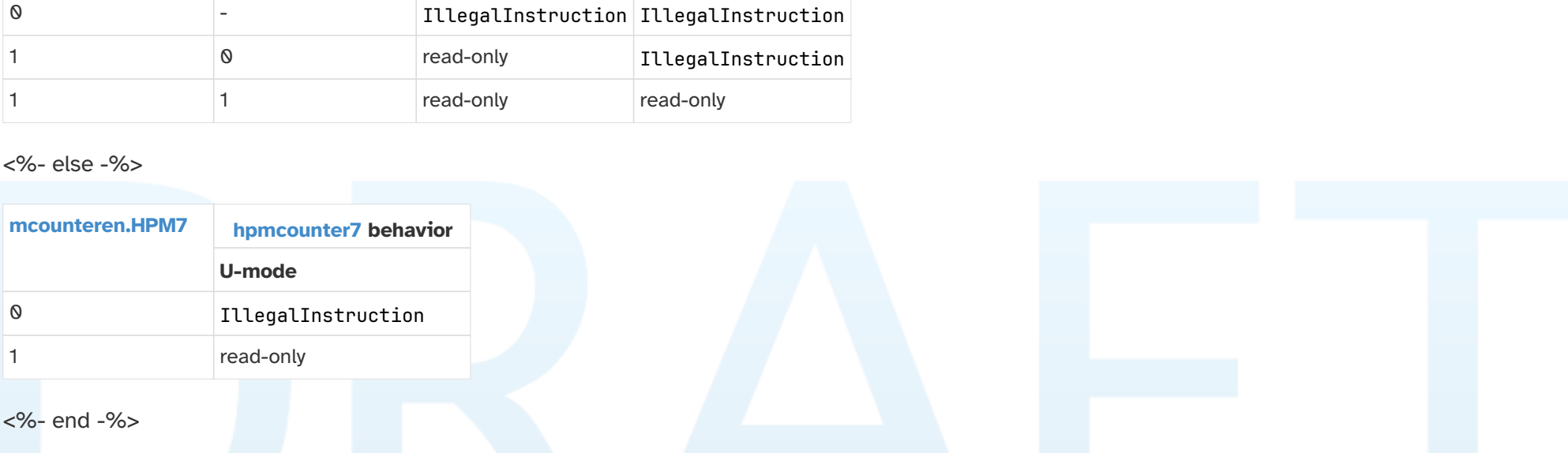



*Figure 32. hpmcounter7 format*

### D.32.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter7.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.32.4. Fields

#### hpmcounter7.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter7.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.32.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM7 & scounteren.HPM7) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM7 == 1'b0 && mcounteren.HPM7 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM7 & scounteren.HPM7) == 1'b0) && (mcounteren.HPM7 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(7);
```

## D.33. hpmcounter8

**User-mode Hardware Performance Counter 5**

Alias for M-mode CSR mhpmcounter8.

Privilege mode access is controlled with mcounteren.HPM8 <%- if ext?(:S) -%> , scounteren.HPM8 <%- if ext?(:H) -%> , and hcounteren.HPM8 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM8 | scounteren.HPM8 | hcounteren.HPM8 | hpmcounter8 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM8 | scounteren.HPM8 | hpmcounter8 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM8 | hpmcounter8 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.33.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc08 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.33.2. Format

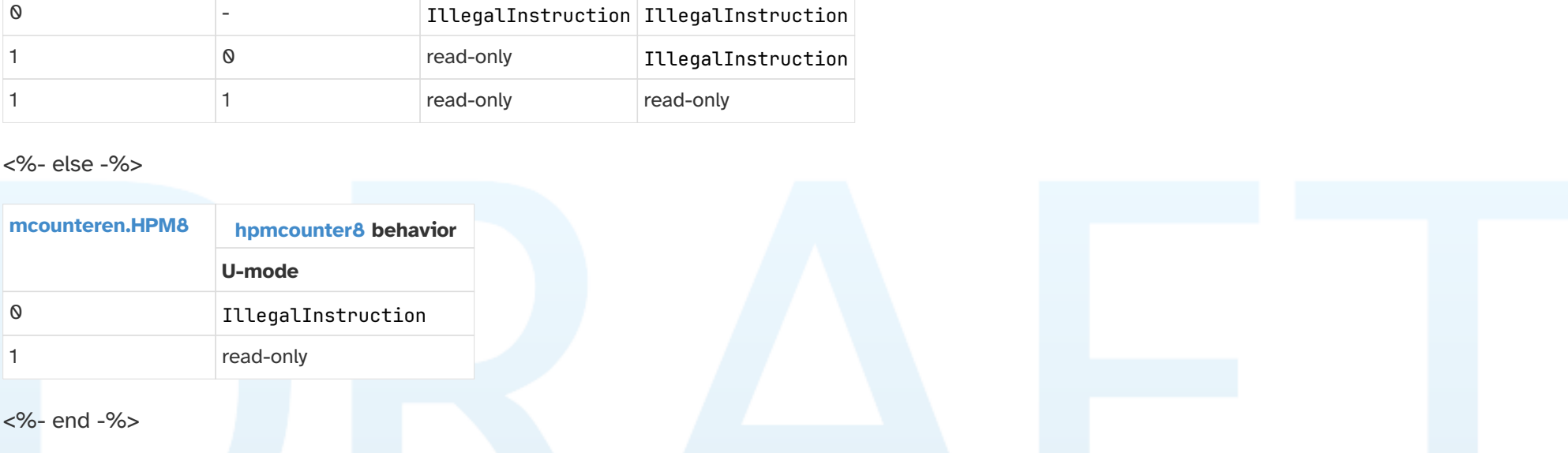



*Figure 33. hpmcounter8 format*

### D.33.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter8.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.33.4. Fields

#### hpmcounter8.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter8.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.33.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM8 & scounteren.HPM8) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM8 == 1'b0 && mcounteren.HPM8 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM8 & scounteren.HPM8) == 1'b0) && (mcounteren.HPM8 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(8);
```

## D.34. hpmcounter9

**User-mode Hardware Performance Counter 6**

Alias for M-mode CSR mhpmcounter9.

Privilege mode access is controlled with mcounteren.HPM9 <%- if ext?(:S) -%> , scounteren.HPM9 <%- if ext?(:H) -%> , and hcounteren.HPM9 <%- end -%> <%- end -%> as follows:

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

| mcounteren.HPM9 | scounteren.HPM9 | hcounteren.HPM9 | hpmcounter9 behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | IllegalInstruction | IllegalInstruction | IllegalInstruction | IllegalInstruction |
| 1 | 0 | 0 | read-only | IllegalInstruction | VirtualInstruction | VirtualInstruction |
| 1 | 1 | 0 | read-only | read-only | VirtualInstruction | VirtualInstruction |
| 1 | 0 | 1 | read-only | IllegalInstruction | read-only | VirtualInstruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

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

| mcounteren.HPM9 | scounteren.HPM9 | hpmcounter9 behavior | |
|---|---|---|---|
| | | S-mode | U-mode |
| 0 | - | IllegalInstruction | IllegalInstruction |
| 1 | 0 | read-only | IllegalInstruction |
| 1 | 1 | read-only | read-only |

<%- else -%>

| mcounteren.HPM9 | hpmcounter9 behavior |
|---|---|
| | U-mode |
| 0 | IllegalInstruction |
| 1 | read-only |

<%- end -%>

### D.34.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc09 |
| **Defining extension** | • Zihpm, version >= Zihpm@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.34.2. Format

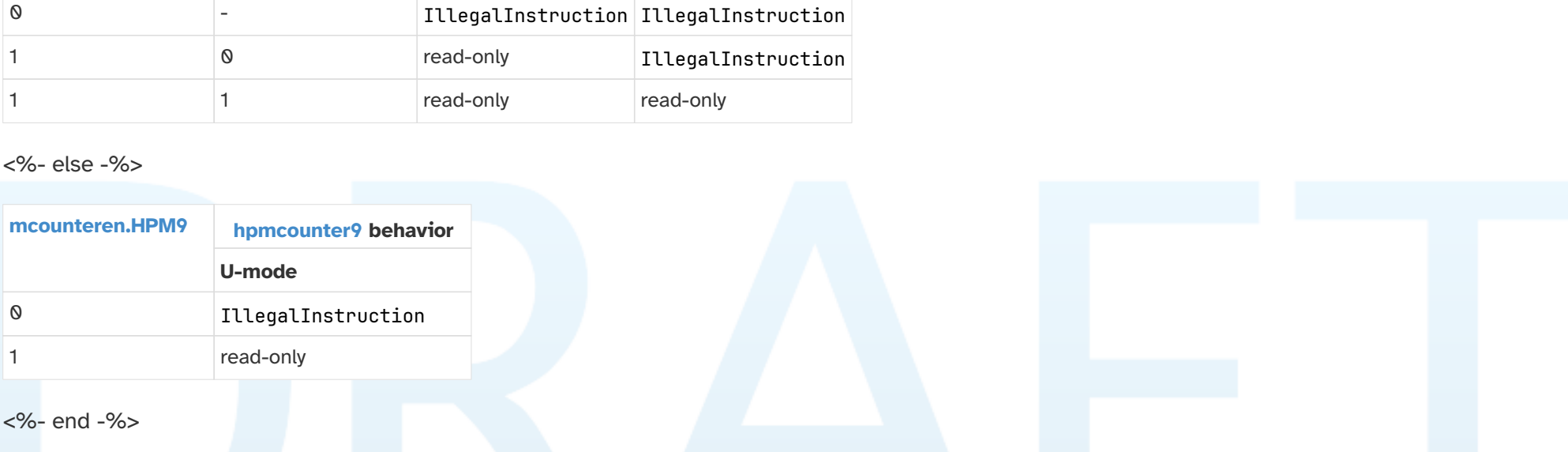



*Figure 34. hpmcounter9 format*

### D.34.3. Field Summary

| Name | Location | Type | Reset Value |
|---|---|---|---|
| hpmcounter9.COUNT | 63:0 | RO-H | UNDEFINED_LEGAL |

### D.34.4. Fields

**hpmcounter9.COUNT Field**

**Location:**

*63:0*

**Description:**

*Alias of mhpmcounter9.COUNT.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.34.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (misa.S == 1'b1) {
    if ((mcounteren.HPM9 & scounteren.HPM9) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (hcounteren.HPM9 == 1'b0 && mcounteren.HPM9 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (hcounteren.HPM9 & scounteren.HPM9) == 1'b0) && (mcounteren.HPM9 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (mcounteren.HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(9);
```

## D.35. instret

### Instructions retired counter for RDINSTRET Instruction

Alias for M-mode CSR minstret.

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

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

### D.35.1. Attributes

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

### D.35.2. Format

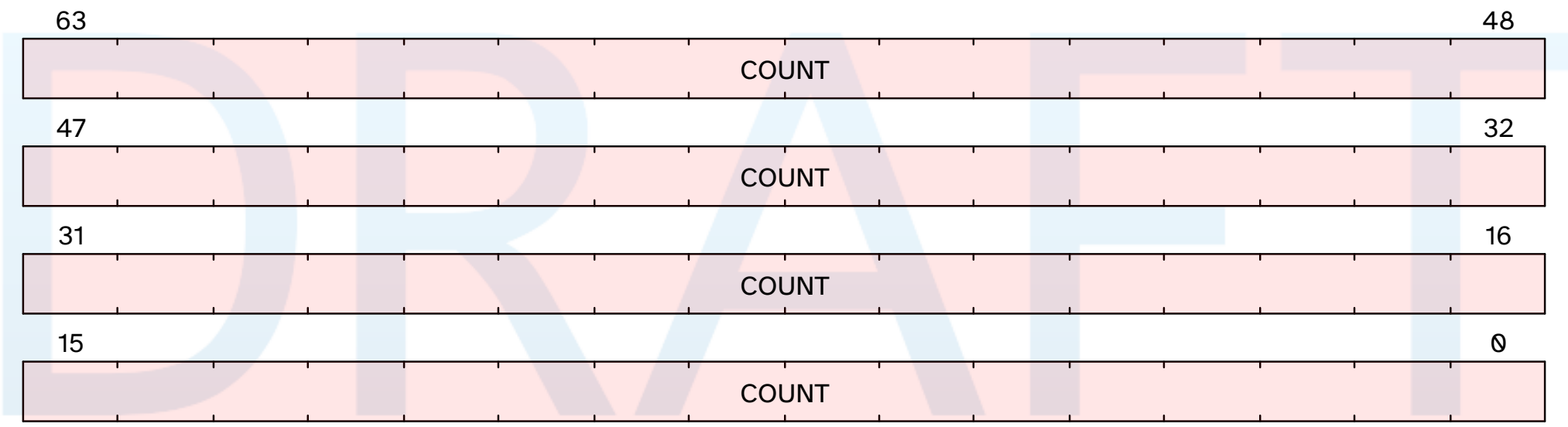



*Figure 35. instret format*

### D.35.3. Field Summary

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

### D.35.4. Fields

#### instret.COUNT Field

**Location:**

*63:0*

**Description:**

*Alias of minstret.COUNT.*

**Type:**

*RO-H*

**Reset value:**

0

### D.35.5. Software read

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

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

## D.36. instreth

### Instructions retired counter, high bits

*instreth is only defined in RV32.*

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

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

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

### D.36.1. Attributes

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

### D.36.2. Format

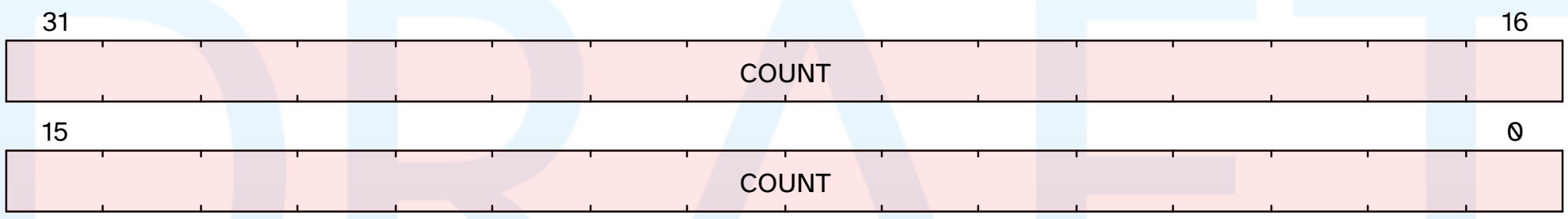



*Figure 36. instreth format*

### D.36.3. Field Summary

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

### D.36.4. Fields

**instreth.COUNT Field**

**Location:**

*31:0*

**Description:**

*Alias of minstret.COUNT[63:32].*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.36.5. Software read

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

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

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

## D.37. mcycle

**Machine Cycle Counter**

Counts the number of clock cycles executed by the processor core on which the hart is running. The counter has 64-bit precision on all RV32 and RV64 harts.

The mcycle CSR may be shared between harts on the same core, in which case writes to mcycle will be visible to those harts. The platform should provide a mechanism to indicate which harts share an mcycle CSR.

### D.37.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xb00 |
| **Defining extension** | • Zicntr, version >= Zicntr@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | M |

### D.37.2. Format

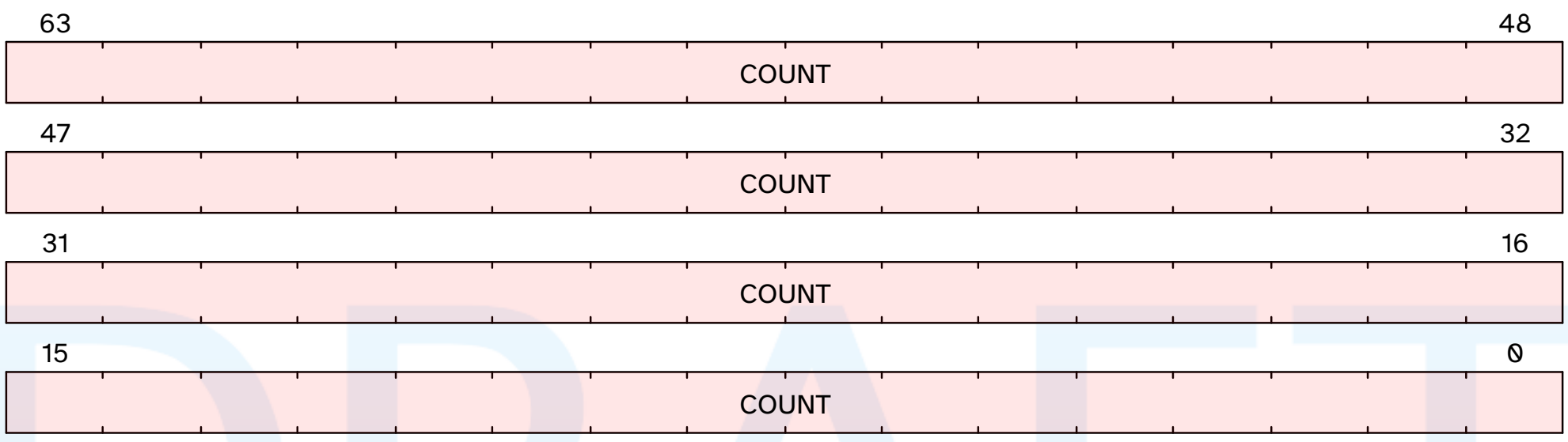



*Figure 37. mcycle format*

### D.37.3. Field Summary

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

### D.37.4. Fields

**mcycle.COUNT Field**

**Location:**

*63:0*

**Description:**

*Cycle counter.*

*<%- if ext?(:Zicntr) -%>*
*Aliased as cycle.*
*<%- end -%>*

*Increments every cycle unless:*

- *mcountinhibit.CY <%- if ext?(:Smcdeleg) -%>or its alias* `scountinhibit.CY` *<%- end -%> is set*
  *<%- if ext?(:Smcntrpmf) -%>*
- `mcyclecfg.MINH` *is set and the current privilege level is M*
  *<%- if ext?(:S) -%>*
- `mcyclecfg.SINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH` *<%- end -%> is set and the current privilege level is (H)S*
  *<%- end -%>*
  *<%- if ext?(:U) -%>*
- `mcyclecfg.UINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH` *<%- end -%> is set and the current privilege level is U*
  *<%- end -%>*

*<%- if ext?(:H) -%>*

- `mcyclecfg.VSINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH`*<%- end -%> is set and the current privilege level is VS*
- `mcyclecfg.VUINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH`*<%- end -%> is set and the current privilege level is VU*

*<%- end -%>*
*<%- end -%>*

**Type:**

*RW-RH*

**Reset value:**

*UNDEFINED_LEGAL*

### D.37.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
  return sw_write_mcycle({read_mcycle()[63:31], csr_value.COUNT[31:0]});
} else {
  return sw_write_mcycle(csr_value.COUNT);
}
```

### D.37.6. Software read

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

```
return read_mcycle();
```

## D.38. mcycleh

### High-half machine Cycle Counter

*mcycleh is only defined in RV32.*

High-half alias of mcycle.

### D.38.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xb80 |
| **Defining extension** | • Zicntr, version >= Zicntr@2.0.0 |
| **Length** | 32-bit |
| **Privilege Mode** | M |

### D.38.2. Format

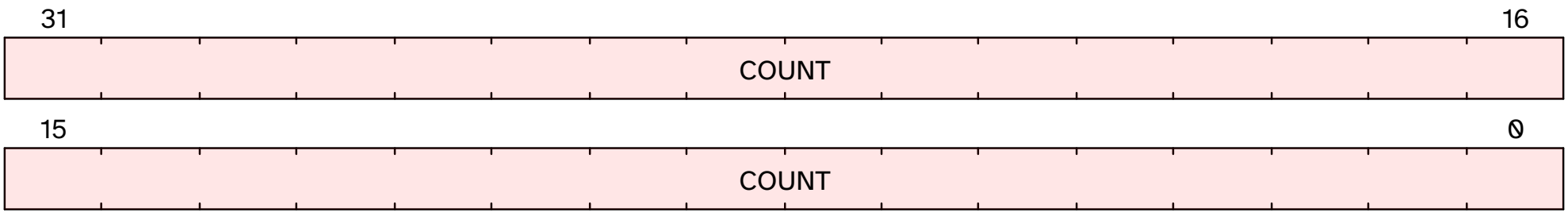



*Figure 38. mcycleh format*

### D.38.3. Field Summary

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

### D.38.4. Fields

**mcycleh.COUNT Field**

**Location:**

*31:0*

**Description:**

*Alias of upper half of mcycle.COUNT.*

**Type:**

*RW-RH*

**Reset value:**

*UNDEFINED_LEGAL*

### D.38.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = # since writes to this register may not be hart-local, it must be handled
# as a special case
if (xlen() == 32) {
  return sw_write_mcycle({csr_value.COUNT[31:0], read_mcycle()[31:0]});
} else {
  return sw_write_mcycle(csr_value.COUNT);
}
```

### D.38.6. Software read

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

```
return read_mcycle()[63:32];
```

## D.39. minstret

### Machine Instructions Retired Counter

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

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

### D.39.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xb02 |
| **Defining extension** | • Zicntr, version >= Zicntr@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | M |

### D.39.2. Format

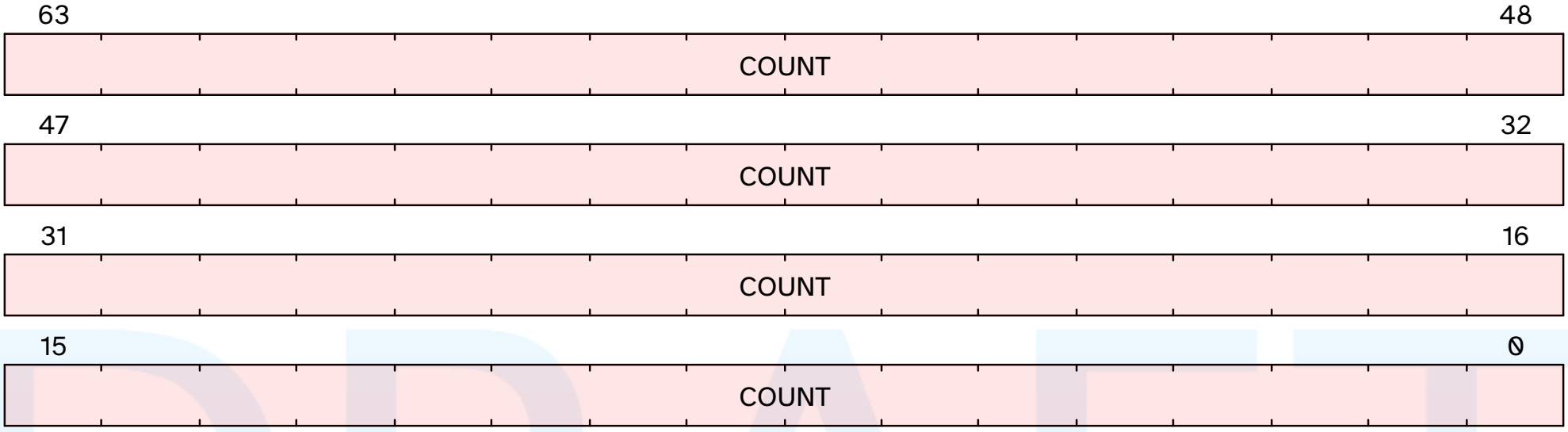



*Figure 39. minstret format*

### D.39.3. Field Summary

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

### D.39.4. Fields

#### minstret.COUNT Field

**Location:**

*63:0*

**Description:**

*Instructions retired counter.*

*<%- if ext?(:Zicntr) -%>*
*Aliased as instret.COUNT.*
*<%- end -%>*

*Increments every time an instruction retires unless:*

- *mcountinhibit.IR <%- if ext?(:Smcdeleg) -%>or its alias* `scountinhibit.IR` *<%- end -%> is set*
  *<%- if ext?(:Smcntrpmf) -%>*
- `minstretcfg.MINH` *is set and the current privilege level is M*
  *<%- if ext?(:S) -%>*
- `minstretcfg.SINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH` *<%- end -%> is set and the current privilege level is (H)S*
  *<%- end -%>*
  *<%- if ext?(:U) -%>*
- `minstretcfg.UINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH` *<%- end -%> is set and the current privilege level is U*
  *<%- end -%>*
  *<%- if ext?(:H) -%>*

- `minstretcfg.VSINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH`*<%- end -%> is set and the current privilege level is VS*
- `minstretcfg.VUINH` *<%- if ext?(:Ssccfg) -%>or its alias* `instretcfg.SINH`*<%- end -%> is set and the current privilege level is VU*

*<%- end -%>*
*<%- end -%>*

*An instruction that causes an exception, notably including MRET/SRET, does not retire and does not cause minstret.COUNT to increment.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

## D.40. minstreth

### Machine Instructions Retired Counter

Upper half of 64-bit instructions retired counters.

See minstret for details.

### D.40.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xb82 |
| **Defining extension** | • Zicntr, version >= Zicntr@2.0.0 |
| **Length** | 32-bit |
| **Privilege Mode** | M |

### D.40.2. Format

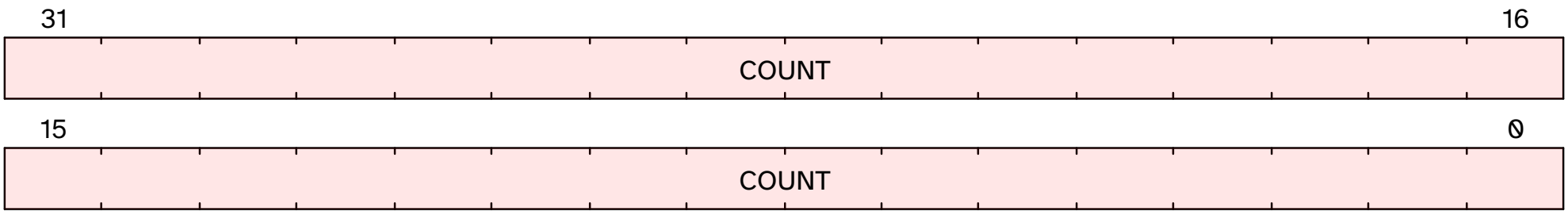



*Figure 40. minstreth format*

### D.40.3. Field Summary

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

### D.40.4. Fields

**minstreth.COUNT Field**

**Location:**

*31:0*

**Description:**

**Instructions retired counter**

*Upper half of minstret.*

**Type:**

*RW-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.40.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

```
COUNT = CSR[mcycle].COUNT = {csr_value.COUNT[31:0], CSR[minstret].COUNT[31:0]};
return csr_value.COUNT;
```

### D.40.6. Software read

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

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

## D.41. time

### Timer for RDTIME Instruction

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR `mtime`.

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

| mcounteren.TM | scounteren.TM | scounteren.TM | time behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | Illegal Instruction | Illegal Instruction | Illegal Instruction | Illegal Instruction |
| 1 | 0 | 0 | read-only | Illegal Instruction | Illegal Instruction | Illegal Instruction |
| 1 | 1 | 0 | read-only | read-only | Illegal Instruction | Illegal Instruction |
| 1 | 0 | 1 | read-only | Illegal Instruction | read-only | Illegal Instruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

### D.41.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc01 |
| **Defining extension** | • Zicntr, version >= Zicntr@2.0.0 |
| **Length** | 64-bit |
| **Privilege Mode** | U |

### D.41.2. Format

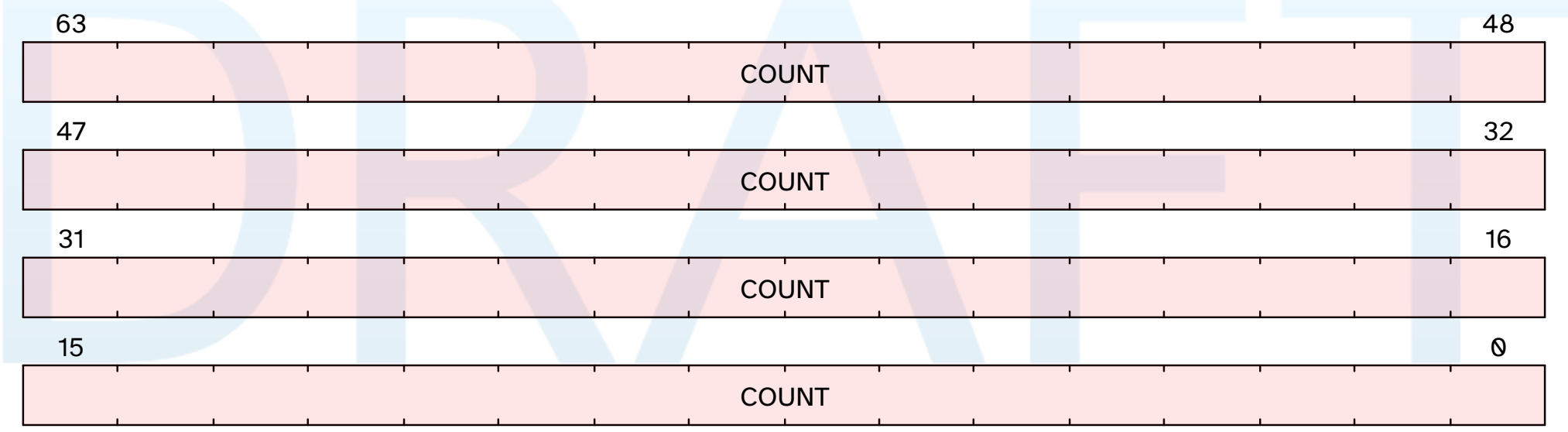



*Figure 41. time format*

### D.41.3. Field Summary

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

### D.41.4. Fields

#### time.COUNT Field

**Location:**

*63:0*

**Description:**

*Reports the current wall-clock time from the timer device.*

*Alias of the `mtime` memory-mapped CSR.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.41.5. Software read

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

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

## D.42. timeh

### High-half timer for RDTIME Instruction

*timeh is only defined in RV32.*

This CSR does not exist, and access will cause an IllegalInstruction exception.

Shadow of the memory-mapped M-mode CSR `mtimeh`.

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

| mcounteren.TM | scounteren.TM | scounteren.TM | time behavior | | | |
|---|---|---|---|---|---|---|
| | | | S-mode | U-mode | VS-mode | VU-mode |
| 0 | - | - | Illegal Instruction | Illegal Instruction | Illegal Instruction | Illegal Instruction |
| 1 | 0 | 0 | read-only | Illegal Instruction | Illegal Instruction | Illegal Instruction |
| 1 | 1 | 0 | read-only | read-only | Illegal Instruction | Illegal Instruction |
| 1 | 0 | 1 | read-only | Illegal Instruction | read-only | Illegal Instruction |
| 1 | 1 | 1 | read-only | read-only | read-only | read-only |

### D.42.1. Attributes

| | |
|---|---|
| **CSR Address** | 0xc81 |
| **Defining extension** | • Zicntr, version >= Zicntr@2.0.0 |
| **Length** | 32-bit |
| **Privilege Mode** | U |

### D.42.2. Format

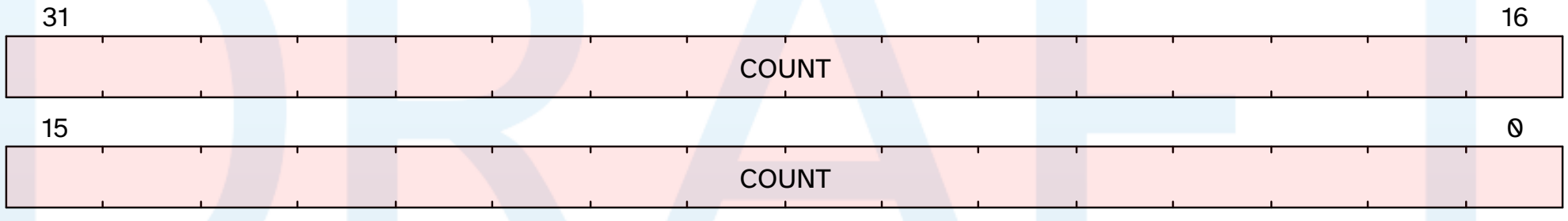



*Figure 42. timeh format*

### D.42.3. Field Summary

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

### D.42.4. Fields

#### timeh.COUNT Field

**Location:**

*31:0*

**Description:**

*Reports the most significant 32 bits of the current wall-clock time from the timer device.*

**Type:**

*RO-H*

**Reset value:**

*UNDEFINED_LEGAL*

### D.42.5. Software read

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

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

# Appendix E: IDL Function Details

## E.1. implemented? (generated)

Return true if the implementation supports `extension`.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `ExtensionName extension` |

## E.2. implemented_version? (generated)

Return true if the implementation supports `extension` meeting 'version_requirement'.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `ExtensionName extension, String version_requirement` |

## E.3. implemented_csr? (generated)

Return true if csr_addr is an implemented CSR

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<12> csr_addr` |

## E.4. direct_csr_lookup (generated)

Return CSR info for a CSR with direct address csr_addr.

If no CSR exists, <return_value>.valid == false

| | |
|---|---|
| **Return Type** | `Csr` |
| **Arguments** | `Bits<12> csr_addr` |

## E.5. indirect_csr_lookup (generated)

Return CSR info for a CSR with indirect address csr_addr at window slot window_slot.

If no CSR exists, <return_value>.valid == false

| | |
|---|---|
| **Return Type** | `Csr` |
| **Arguments** | `Bits<MXLEN> csr_addr, Bits<4> window_slot` |

## E.6. csr_hw_read (generated)

Returns the raw value of csr

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Csr csr` |

### E.7. csr_sw_read (generated)

Returns the result of CSR[csr].sw_read(); i.e., the software view of the register

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Csr csr` |

### E.8. csr_sw_write (generated)

Writes value to csr, applying an WARL transformations first.

Uses the sw_write(...) functions of CSR field definitions.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Csr csr, Bits<MXLEN> value` |

### E.9. unpredictable (builtin)

Indicate that the hart has reached a state that is unpredictable because the RISC-V spec allows multiple behaviors. Generally, this will be a fatal condition to any emulation, since it is unclear what to do next.

The single argument *why* is a string describing why the hart entered an unpredictable state.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `String why` |

### E.10. read_hpm_counter (builtin)

Returns the value of hpmcounterN.

N must be between 3..31.

hpmcounterN must be implemented.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<5> n` |

### E.11. hartid (builtin)

Returns the value for mhartid as seen by this hart.

Must obey the rules of the priv spec:

> The mhartid CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | None |

### E.12. read_mcycle (builtin)

Return the current value of the cycle counter.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | None |

### E.13. read_mtime (builtin)

Return the current value of the real time device.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | None |

### E.14. sw_write_mcycle (builtin)

Given a *value* that software is trying to write into mcycle, perform the write and return the value that will actually be written.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<64> value` |

### E.15. cache_block_zero (builtin)

Zero the cache block at the given physical address.

The cache block may be zeroed using 1 or more writes.

A cache-block-sized region is zeroed regardless of whether or not the memory is in a cacheable PMA region.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg cache_block_physical_address` |

### E.16. eei_ecall_from_m (builtin)

When TRAP_ON_ECALL_FROM_M is false, this function will be called to emulate the EEI handling of ECALL-from-M.

If TRAP_ON_ECALL_FROM_M is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

### E.17. eei_ecall_from_s (builtin)

When TRAP_ON_ECALL_FROM_S is false, this function will be called to emulate the EEI handling of ECALL-from-S.

If TRAP_ON_ECALL_FROM_S is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

### E.18. eei_ecall_from_u (builtin)

When TRAP_ON_ECALL_FROM_U is false, this function will be called to emulate the EEI handling of ECALL-from-U.

If TRAP_ON_ECALL_FROM_U is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

| | |
|---|---|
| **Return Type** | `void` |

| | |
|---|---|
| **Arguments** | None |

## E.19. eei_ecall_from_vs (builtin)

When TRAP_ON_ECALL_FROM_VS is false, this function will be called to emulate the EEI handling of ECALL-from-VS.

If TRAP_ON_ECALL_FROM_VS is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.20. eei_ebreak (builtin)

When TRAP_ON_EBREAK is false, this function will be called to emulate the EEI handling of EBREAK

If TRAP_ON_EBREAK is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.21. memory_model_acquire (builtin)

Perform an acquire; that is, ensure that no subsequent operation in program order appears to an external observer to occur after the operation calling this function.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.22. memory_model_release (builtin)

Perform a release; that is, ensure that no prior store in program order can be observed external to this hart after this function returns.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.23. assert (builtin)

Assert that a condition is true. Failure represents an error in the IDL model.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Boolean test, String message` |

## E.24. notify_mode_change (builtin)

Called whenever the privilege mode changes. Downstream tools can use this to hook events.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode new_mode, PrivilegeMode old_mode` |

## E.25. abort_current_instruction (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

### E.26. ebreak (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

### E.27. prefetch_instruction (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address` |

### E.28. prefetch_read (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address` |

### E.29. prefetch_write (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address` |

### E.30. fence (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Boolean pi, Boolean pr, Boolean po, Boolean pw, Boolean si, Boolean sr, Boolean so, Boolean sw` |

### E.31. fence_tso (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.32. ifence (builtin)

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

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.33. pause (builtin)

Pause hart retirement for a implementation-defined period of time, which may be zero.

See Zihintpause for more.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.34. pow (generated)

Return `value` to the power `exponent`.

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `XReg value, XReg exponent` |

## E.35. maybe_cache_translation (generated)

Given a translation result, potentially cache the result for later use. This function models a TLB fill operation. A valid implementation does nothing.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg vaddr, MemoryOperation op, TranslationResult result` |

## E.36. cached_translation (generated)

Possibly returns a cached translation result matching vaddr.

CachedTranslationResult contains a Boolean 'valid' field. If valid, 'result' is a usable translation. Otherwise, the cache lookup failed.

| | |
|---|---|
| **Return Type** | `CachedTranslationResult` |
| **Arguments** | `XReg vaddr, MemoryOperation op` |

## E.37. order_pgtbl_writes_before_vmafence (builtin)

Orders all writes prior to this call in global memory order that affect a page table in the set identified by order_type before any subsequent sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.W.INVAL.

A valid implementation does nothing if address caching is not used.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `VmaOrderType order_type` |

## E.38. order_pgtbl_reads_after_vmafence (builtin)

Orders all reads after to this call in global memory order to a page table in the set identified by order_type after any prior sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.INVAL.IR.

A valid implementation does nothing if address caching is not used.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `VmaOrderType order_type` |

## E.39. invalidate_translations (generated)

Locally invalidate the cached S-mode/VS-mode/G-stage address translations contained in the set identified by inval_type.

A valid implementation does nothing if address caching is not used.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `VmaOrderType inval_type` |

## E.40. read_physical_memory

Read from physical memory.

| | |
|---|---|
| **Return Type** | `Bits<len>` |
| **Arguments** | `XReg paddr` |

```
if (len == 8) {
  return read_physical_memory_8(paddr);
} else if (len == 16) {
  return read_physical_memory_16(paddr);
} else if (len == 32) {
  return read_physical_memory_32(paddr);
} else if (len == 64) {
  return read_physical_memory_64(paddr);
} else {
  assert(false, "Invalid len");
}
```

## E.41. read_physical_memory_8 (builtin)

Read a byte from physical memory.

| | |
|---|---|
| **Return Type** | `Bits⑧` |
| **Arguments** | `XReg paddr` |

## E.42. read_physical_memory_16 (builtin)

Read two bytes from physical memory.

| | |
|---|---|
| **Return Type** | `Bits⑯` |

| | |
|---|---|
| **Arguments** | `XReg paddr` |

## E.43. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `XReg paddr` |

## E.44. read_physical_memory_64 (builtin)

Read eight bytes from physical memory.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `XReg paddr` |

## E.45. write_physical_memory

Write to physical memory.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg paddr, Bits<len> value` |

```
if (len == 8) {
  write_physical_memory_8(paddr, value);
} else if (len == 16) {
  write_physical_memory_16(paddr, value);
} else if (len == 32) {
  write_physical_memory_32(paddr, value);
} else if (len == 64) {
  write_physical_memory_64(paddr, value);
} else {
  assert(false, "Invalid len");
}
```

## E.46. write_physical_memory_8 (builtin)

Write a byte to physical memory.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg paddr, Bits<8> value` |

## E.47. write_physical_memory_16 (builtin)

Write two bytes to physical memory.

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

## E.48. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

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

## E.49. write_physical_memory_64 (builtin)

Write eight bytes to physical memory.

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

## E.50. wfi (builtin)

Wait-for-interrupt: hint that the processor should enter a low power state until the next interrupt.

A valid implementation is a no-op.

The model will advance the PC; this function does not need to.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

## E.51. pma_applies? (builtin)

Checks if *attr* is applied to the entire physical address region between [paddr, paddr + len) based on static PMA attributes.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `PmaAttribute attr, Bits<PHYS_ADDR_WIDTH> paddr, U32 len` |

## E.52. atomic_check_then_write_32 (builtin)

Atomically:

- Reads 32-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 32-bits

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, Bits<32> compare_value, Bits<32> write_value` |

## E.53. atomic_check_then_write_64 (builtin)

Atomically:

- Reads 64-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 64-bits

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, Bits<64> compare_value, Bits<64> write_value` |

## E.54. atomically_set_pte_a (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' bit and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len` |

## E.55. atomically_set_pte_a_d (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' and 'D' bits and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len` |

## E.56. atomic_read_modify_write_32 (builtin)

Atomically read-modify-write 32-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

| | |
|---|---|
| **Return Type** | `Bits` |

| | |
|---|---|
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<32> value, AmoOperation op` |

## E.57. atomic_read_modify_write_64 (builtin)

Atomically read-modify-write 64-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<64> value, AmoOperation op` |

## E.58. set_external_interrupt

Set an external interrupt targeting target_mode

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode target_mode` |

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

## E.59. clear_external_interrupt

Clear an external interrupt targeting target_mode

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode target_mode` |

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

## E.60. set_software_interrupt

Set a software interrupt targeting target_mode

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode target_mode` |

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

### E.61. clear_software_interrupt

Clear a software interrupt targeting target_mode

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode target_mode` |

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

### E.62. set_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode target_mode` |

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

### E.63. clear_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode target_mode` |

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

### E.64. refresh_pending_interrupts

refreshes the calculation of a pending interrupt

needs to be called after any state update that could change a pending interrupt. This includes: - CSR[mip] - CSR[mie] - CSR[mstatus].MIE - CSR[mstatus].SIE - CSR[vsstatus].SIE - CSR[mideleg] - CSR[sideleg] - CSR[hideleg] - CSR[hvip] - CSR[hgeip] - CSR[hgeie] - mode changes

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

```
Bits<MXLEN> pending_ints = mip.sw_read() & $bits(mie);
if (pending_ints == 0) {
  pending_and_enabled_interrupts = 0;
  return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, ">
1.9.1") && implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_enabled_ints = mode() == PrivilegeMode::M) && (mstatus.MIE == 1'b0 ? 0 : ($bits(mie) & (HAS_MIDELEG ?
~$bits(mideleg) : ~MXLEN'0));
Bits<MXLEN> mmode_pending_and_enabled = pending_ints & mmode_enabled_ints;
if (mmode_pending_and_enabled != 0) {
  pending_and_enabled_interrupts = mmode_pending_and_enabled;
  return ;
}
if (misa.S == 1'b1) {
  Bits<MXLEN> smode_enabled_ints = mode() == PrivilegeMode::M) || (mstatus.SIE == 1'b0 ? 0 : $bits(mie) &
($bits(mideleg));
  Bits<MXLEN> smode_pending_and_enabled = pending_ints & smode_enabled_ints;
  if (smode_pending_and_enabled != 0) {
    pending_and_enabled_interrupts = smode_pending_and_enabled;
    return ;
  }
}
pending_and_enabled_interrupts = 0;
```

### E.65. highest_priority_interrupt

Given a bitmask of interrupts in the format of MIE/MIP, return the highest priority interrupt code that is set

Interrupt priority is: MEI, MSI, MTI, SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI

| | |
|---|---|
| **Return Type** | `InterruptCode` |
| **Arguments** | `Bits<MXLEN> int_mask` |

```
if (int_mask[$bits(InterruptCode::MachineExternal)] == 1'b1) {
  return InterruptCode::MachineExternal;
} else if (int_mask[$bits(InterruptCode::MachineSoftware)] == 1'b1) {
  return InterruptCode::MachineSoftware;
} else if (int_mask[$bits(InterruptCode::MachineTimer)] == 1'b1) {
  return InterruptCode::MachineTimer;
}
if (misa.S == 1'b1) {
  if (int_mask[$bits(InterruptCode::SupervisorExternal)] == 1'b1) {
```

```
    return InterruptCode::SupervisorExternal;
  } else if (int_mask[$bits(InterruptCode::SupervisorSoftware)] == 1'b1) {
    return InterruptCode::SupervisorSoftware;
  } else if (int_mask[$bits(InterruptCode::SupervisorTimer)] == 1'b1) {
    return InterruptCode::SupervisorTimer;
  }
}
if (implemented?(ExtensionName::Sscofpmf)) {
  if (int_mask[$bits(InterruptCode::LocalCounterOverflow)] == 1'b1) {
    return InterruptCode::LocalCounterOverflow;
  }
}
assert(false, "There is no valid interrupt");
```

## E.66. choose_interrupt

Return the highest priority interrupt that is both pending and enabled and the mode it will be taken in

| | |
|---|---|
| **Return Type** | `InterruptCode, PrivilegeMode` |
| **Arguments** | None |

```
InterruptCode chosen;
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, ">
1.9.1") && implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_pending_and_enabled = pending_and_enabled_interrupts & ~(HAS_MIDELEG ? $bits(mideleg) : MXLEN'0);
if (mmode_pending_and_enabled != 0) {
  assert((mode() != PrivilegeMode::M) || (mstatus.MIE == 1'b1), "M-mode interrupts are not enabled");
  chosen = highest_priority_interrupt(mmode_pending_and_enabled);
} else if (misa.S == 1'b1) {
  Bits<MXLEN> smode_pending_and_enabled = (pending_and_enabled_interrupts & $bits(mideleg));
  if (smode_pending_and_enabled != 0) {
    assert((mode() == PrivilegeMode::U) || (mode() == PrivilegeMode::VU) || (mode() == PrivilegeMode::VS) || (mode() ==
PrivilegeMode::S) && (mstatus.SIE == 1'b1), "S-mode interrupt can't be triggered");
    chosen = highest_priority_interrupt(smode_pending_and_enabled);
  }
}
assert($bits(chosen) != 0, "Didn't pick interrupt?");
PrivilegeMode to_mode;
Bits<MXLEN> chosen_mask = (MXLEN'1 << $bits(chosen));
if (((HAS_MIDELEG ? $bits(mideleg) : MXLEN'0) & chosen_mask) == 0) {
  to_mode = PrivilegeMode::M;
} else {
  if (misa.S == 1'b1) {
    to_mode = PrivilegeMode::S;
  } else {
    to_mode = PrivilegeMode::U;
  }
}
return chosen, to_mode;
```

## E.67. take_interrupt

Take (adjust CSRs and set PC to handler) the highest priority interrupt that is both pending and enabled

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

```
PrivilegeMode to_mode;
InterruptCode code;
(code, to_mode = choose_interrupt());
if (to_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
  mstatus.MPP = $bits(mode())[1:0];
  if (misa.H == 1'b1) {
    mstatus.MPV = $bits(mode())[2];
    mtval2.VALUE = 0;
```

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

## E.68. fetch_memory_aligned_16

Fetch 16 bits from virtual memory using a known aligned address.

| | |
|---|---|
| **Return Type** | `Bits⑯` |
| **Arguments** | `XReg virtual_address` |

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 16, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<16>(result.paddr);
```

## E.69. fetch_memory_aligned_32

Fetch 32 bits from virtual memory using a known aligned address.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `XReg virtual_address` |

```
TranslationResult result;
```

```
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 32, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<32>(result.paddr);
```

### E.70. power_of_2?

Returns true if value is a power of two, false otherwise

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<N> value` |

```
return (value != 0) && value & (value - 1 == 0);
```

### E.71. ary_includes?

Returns true if *value* is an element of ary, and false otherwise

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<ELEMENT_SIZE> ary[ARY_SIZE], Bits<ELEMENT_SIZE> value` |

```
for (U32 i = 0; i < ARY_SIZE; i++) {
  if (ary[i] == value) {
    return true;
  }
}
return false;
```

### E.72. has_virt_mem?

Returns true if some virtual memory translation (Sv*) is supported in the config.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
return implemented?(ExtensionName::Sv32) || implemented?(ExtensionName::Sv39) || implemented?(ExtensionName::Sv48) ||
implemented?(ExtensionName::Sv57);
```

### E.73. highest_set_bit

Returns the position of the highest (nearest MSB) bit that is '1', or -1 if value is zero.

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `XReg value` |

```
for (U32 i = xlen() - 1; i >= 0; i--) {
  if (value[i] == 1) {
    return i;
  }
```

```
}
return -'sd1;
```

### E.74. lowest_set_bit

Returns the position of the lowest (nearest LSB) bit that is '1', or XLEN if value is zero.

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `XReg value` |

```
for (U32 i = 0; i < xlen(); i++) {
  if (value[i] == 1) {
    return i;
  }
}
return xlen();
```

### E.75. bit_length

Returns the minimum number of bits needed to represent value.

Only works on unsigned values.

The value 0 returns 1.

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `XReg value` |

```
for (XReg i = 63; i > 0; i--) {
  if (value[i] == 1) {
    return i;
  }
}
return 1;
```

### E.76. count_leading_zeros

Returns the number of leading 0 bits before the most-significant 1 bit of value, or N if value is zero.

| | |
|---|---|
| **Return Type** | `Bits<bit_length(N)>` |
| **Arguments** | `Bits<N> value` |

```
for (U32 i = 0; i < N; i++) {
  if (value[N - 1 - i] == 1) {
    return i;
  }
}
return N;
```

### E.77. sext

Sign extend `value` starting at `first_extended_bit`.

Bits [`XLEN-1`:`first_extended_bit`] of the return value should get the value of bit (`first_extended bit - 1`).

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `XReg value, XReg first_extended_bit` |

```
if (first_extended_bit == MXLEN) {
  return value;
} else {
  Bits<1> sign = value[first_extended_bit - 1];
  for (U32 i = MXLEN - 1; i >= first_extended_bit; i--) {
    value[i] = sign;
  }
  return value;
}
```

## E.78. is_naturally_aligned

Checks if value is naturally aligned to N bits.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg value` |

```
return true if (N == 8);
XReg Mask = (N / 8) - 1;
return (value & ~Mask) == value;
```

## E.79. in_naturally_aligned_region?

Checks if a length-bit access starting at address lies entirely within an N-bit naturally-aligned region.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg address, U32  length` |

```
XReg Mask = (N / 8) - 1;
return (address & ~Mask) == ((address + length - 1) & ~Mask);
```

## E.80. contains?

Given a *region* defined by region_start, region_size, determine if a *target* defined by target_start, target_size is completely contained with the region.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg region_start, U32  region_size, XReg target_start, U32  target_size` |

```
return target_start >= region_start && (target_start + target_size) <= (region_start + region_size);
```

## E.81. set_fp_flag

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

| | |
|---|---|
| **Return Type** | `void` |

| | |
|---|---|
| **Arguments** | `FpFlag flag` |

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

## E.82. rm_to_mode

Convert rm to a RoundingMode.

encoding is the full encoding of the instruction rm comes from.

Will raise an IllegalInstruction exception if rm is a reserved encoding.

| | |
|---|---|
| **Return Type** | `RoundingMode` |
| **Arguments** | `Bits<3> rm, Bits<32> encoding` |

```
if (rm == $bits(RoundingMode::RNE)) {
  return RoundingMode::RNE;
} else if (rm == $bits(RoundingMode::RTZ)) {
  return RoundingMode::RTZ;
} else if (rm == $bits(RoundingMode::RDN)) {
  return RoundingMode::RDN;
} else if (rm == $bits(RoundingMode::RUP)) {
  return RoundingMode::RUP;
} else if (rm == $bits(RoundingMode::RMM)) {
  return RoundingMode::RMM;
} else if (rm == $bits(RoundingMode::DYN)) {
  return $enum(RoundingMode, fcsr.FRM);
} else {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

## E.83. mark_f_state_dirty

Potentially updates mstatus.FS to the Dirty (3) state, depending on configuration settings.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

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

## E.84. nan_box

Produces a properly NaN-boxed floating-point value from a floating-point value of smaller size by adding all 1's to the upper bits.

| | |
|---|---|
| **Return Type** | `Bits<TO_SIZE>` |
| **Arguments** | `Bits<FROM_SIZE> from_value` |

```
assert(FROM_SIZE < TO_SIZE, "Bad template arguments; FROM_SIZE must be less than TO_SIZE");
return {{TO_SIZE - FROM_SIZE{1'b1}}, from_value};
```

## E.85. check_f_ok

Checks if instructions from the F extension can be executed, and, if not, raise an exception.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Bits<INSTR_ENC_SIZE> encoding` |

```
if (MUTABLE_MISA_F && misa.F == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
if (mstatus.FS == 0) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

## E.86. is_sp_neg_inf?

Return true if sp_value is negative infinity.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return sp_value == SP_NEG_INF;
```

## E.87. is_sp_pos_inf?

Return true if sp_value is positive infinity.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return sp_value == SP_POS_INF;
```

## E.88. is_sp_neg_norm?

Returns true if sp_value is a negative normal number.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[31] == 1) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

### E.89. is_sp_pos_norm?

Returns true if sp_value is a positive normal number.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[31] == 0) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

### E.90. is_sp_neg_subnorm?

Returns true if sp_value is a negative subnormal number.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[31] == 1) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

### E.91. is_sp_pos_subnorm?

Returns true if sp_value is a positive subnormal number.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[31] == 0) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

### E.92. is_sp_neg_zero?

Returns true if sp_value is negative zero.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return sp_value == SP_NEG_ZERO;
```

### E.93. is_sp_pos_zero?

Returns true if sp_value is positive zero.

| | |
|---|---|
| **Return Type** | `Boolean` |

| | |
|---|---|
| **Arguments** | `Bits<32> sp_value` |

```
return sp_value == SP_POS_ZERO;
```

### E.94. is_sp_nan?

Returns true if sp_value is a NaN (quiet or signaling)

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22:0] != 0);
```

### E.95. is_sp_signaling_nan?

Returns true if sp_value is a signaling NaN

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 0) && (sp_value[21:0] != 0);
```

### E.96. is_sp_quiet_nan?

Returns true if sp_value is a quiet NaN

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<32> sp_value` |

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 1);
```

### E.97. softfloat_shiftRightJam32

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned. The value of dist can be arbitrarily large. In particular, if dist is greater than 32, the result will be either 0 or 1, depending on whether a is zero or nonzero.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<32> a, Bits<32> dist` |

```
return (dist < 31) ? a >> dist | (a << (-dist & 31 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

### E.98. softfloat_shiftRightJam64

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned.

The value of 'dist' can be arbitrarily large. In particular, if dist is greater than 64, the result will be either 0 or 1, depending on whether a is zero or nonzero.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<64> a, Bits<32> dist` |

```
return (dist < 63) ? a >> dist | (a << (-dist & 63 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

## E.99. softfloat_roundToI32

Round to unsigned 32-bit integer, using rounding_mode

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<1> sign, Bits<64> sig, RoundingMode roundingMode` |

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
    roundIncrement = 0xFFF;
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 0) != (sign == 1) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

## E.100. packToF32UI

Pack components into a 32-bit value

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<1> sign, Bits<8> exp, Bits<23> sig` |

```
return {sign, exp, sig};
```

## E.101. packToF16UI

Pack components into a 16-bit value

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<1> sign, Bits<5> exp, Bits<10> sig` |

```
return {sign, exp, sig};
```

## E.102. softfloat_normSubnormalF16Sig

normalize subnormal half-precision value

| | |
|---|---|
| **Return Type** | `Bits<5>, Bits⑩` |
| **Arguments** | `Bits<16> hp_value` |

```
Bits<8> shift_dist = count_leading_zeros<16>(hp_value);
return 1 - shift_dist, hp_value << shift_dist;
```

## E.103. softfloat_roundPackToF32

Round FP value according to mdode and then pack it in IEEE format.

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode` |

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
  roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
  if ($signed(exp) < 0) {
    Boolean isTiny = ($signed(exp) < -8's1) || (sig + roundIncrement < 0x80000000);
    sig = softfloat_shiftRightJam32(sig, -exp);
    exp = 0;
    roundBits = sig & 0x7F;
    if (isTiny && (roundBits != 0)) {
      set_fp_flag(FpFlag::UF);
    }
  } else if (0xFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
    set_fp_flag(FpFlag::OF);
    set_fp_flag(FpFlag::NX);
    return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0) ? 1 : 0);   } } sig = (sig + roundIncrement);
if (sig == 0) {
  exp = 0;
}
return packToF32UI(sign, exp, sig);
```

## E.104. softfloat_normRoundPackToF32

Normalize, round, and pack into a 32-bit floating point value

| | |
|---|---|
| **Return Type** | `Bits` |
| **Arguments** | `Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode` |

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
  return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
  return softfloat_roundPackToF32(sign, exp, sig << shiftDist, mode);
}
```

### E.105. signF32UI

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

| | |
|---|---|
| **Return Type** | Bits① |
| **Arguments** | Bits<32> a |

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

### E.106. expF32UI

Extract exponent of a 32-bit floating point number

| | |
|---|---|
| **Return Type** | Bits⑧ |
| **Arguments** | Bits<32> a |

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

### E.107. fracF32UI

Extract significand of a 32-bit floating point number

| | |
|---|---|
| **Return Type** | Bits |
| **Arguments** | Bits<32> a |

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

### E.108. returnNonSignalingNaN

Returns a non-signalling NaN version of the floating-point number Does not modify the input

| | |
|---|---|
| **Return Type** | U32 |
| **Arguments** | U32 a |

```
U32 a_copy = a;
a_copy[22] = 1'b1;
return a_copy;
```

### E.109. returnMag

Returns magnitude of the given number Does not modify the input

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a` |

```
U32 a_copy = a;
a_copy[31] = 1'b0;
return a_copy;
```

### E.110. returnLargerMag

Returns the larger number between a and b by magnitude If either number is signaling NaN then that is made quiet

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, U32 b` |

```
U32 mag_a = returnMag(a);
U32 mag_b = returnMag(b);
U32 nonsig_a = returnNonSignalingNaN(a);
U32 nonsig_b = returnNonSignalingNaN(b);
if (mag_a < mag_b) {
  return nonsig_b;
}
if (mag_b < mag_a) {
  return nonsig_a;
}
return (nonsig_a < nonsig_b) ? nonsig_a : nonsig_b;
```

### E.111. softfloat_propagateNaNF32UI

Interpreting 'a' and 'b' as the bit patterns of two 32-bit floating- | point values, at least one of which is a NaN, returns the bit pattern of | the combined NaN result. If either 'a' or 'b' has the pattern of a | signaling NaN, the invalid exception is raised.

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, U32 b` |

```
Boolean isSigNaN_a = is_sp_signaling_nan?(a);
Boolean isSigNaN_b = is_sp_signaling_nan?(b);
if (isSigNaN_a || isSigNaN_b) {
  set_fp_flag(FpFlag::NV);
}
return SP_CANONICAL_NAN;
```

### E.112. softfloat_addMagsF32

Returns sum of the magnitudes of 2 floating point numbers

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, U32 b, RoundingMode mode` |

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'd0) {
    z = a + b;
    return z;
  }
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  signZ = signF32UI(a);
  expZ = expA;
  sigZ = 32'h01000000 + sigA + sigB;
  if (sigZ & 0x1) == 0) && (expZ < 8'hFE {
    sigZ = sigZ >> 1;
    return (32'h0 + (signZ << 31) + (expZ << 23) + sigZ);
  }
  sigZ = sigZ << 6;
} else {
  signZ = signF32UI(a);
  U32 sigA_32 = 32'h0 + (sigA << 6);
  U32 sigB_32 = 32'h0 + (sigA << 6);
  if (expDiff < 0) {
    if (expB == 8'hFF) {
      if (sigB != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, 8'hFF, 23'h0);
    }
    expZ = expB;
    sigA_32 = (expA == 0) ? 2 * sigA_32 : (sigA_32 + 0x20000000);
    sigA_32 = softfloat_shiftRightJam32(sigA_32, (32'h0 - expDiff));
  } else {
    if (expA == 8'hFF) {
      if (sigA != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
    }
    expZ = expA;
    sigB_32 = (expB == 0) ? 2 * sigB_32 : (sigB_32 + 0x20000000);
    sigB_32 = softfloat_shiftRightJam32(sigB_32, (32'h0 + expDiff));
  }
  U32 sigZ = 0x20000000 + sigA + sigB;
  if (sigZ < 0x40000000) {
    expZ = expZ - 1;
    sigZ = sigZ << 1;
  }
}
return softfloat_roundPackToF32(signZ, expZ, sigZ[22:0], mode);
```

## E.113. softfloat_subMagsF32

Returns difference of the magnitudes of 2 floating point numbers

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, U32 b, RoundingMode mode` |

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
U32 sigDiff;
U32 sigX;
U32 sigY;
U32 sigA_32;
U32 sigB_32;
Bits<8> shiftDist;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  sigDiff = sigA - sigB;
  if (sigDiff == 0) {
    return packToF32UI(((mode == RoundingMode::RDN) ? 1 : 0), 0, 0);
  }
  if (expA != 0) {
    expA = expA - 1;
  }
  signZ = signF32UI(a);
  if (sigDiff < 0) {
    signZ = ~signZ;
    sigDiff = -32'sh1 * sigDiff;
  }
  shiftDist = count_leading_zeros<32>(sigDiff) - 8;
  expZ = expA - shiftDist;
  if (expZ < 0) {
    shiftDist = expA;
    expZ = 0;
  }
  return packToF32UI(signZ, expZ, sigDiff << shiftDist);
} else {
  signZ = signF32UI(a);
  sigA_32 = 32'h0 + (sigA << 7);
  sigB_32 = 32'h0 + (sigB << 7);
  if (expDiff < 0) {
    signZ = ~signZ;
    if (expB == 0xFF) {
      if (sigB_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, expB, 0);
    }
    expZ = expB - 1;
    sigX = sigB_32 | 0x40000000;
    sigY = sigA_32 + ((expA != 0) ? 0x40000000 : sigA_32);
    expDiff = -expDiff;
  } else {
    if (expA == 0xFF) {
      if (sigA_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
    }
    expZ = expA - 1;
    sigX = sigA_32 | 0x40000000;
    sigY = sigB_32 + ((expB != 0) ? 0x40000000 : sigB_32);
  }
  return softfloat_normRoundPackToF32(signZ, expZ, sigX - softfloat_shiftRightJam32(sigY, expDiff), mode);
}
```

## E.114. f32_add

Returns sum of 2 floating point numbers

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, U32 b, RoundingMode mode` |

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_subMagsF32(a, b, mode);
} else {
  return softfloat_addMagsF32(a, b, mode);
}
```

### E.115. f32_sub

Returns difference of 2 floating point numbers

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, U32 b, RoundingMode mode` |

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_addMagsF32(a, b, mode);
} else {
  return softfloat_subMagsF32(a, b, mode);
}
```

### E.116. i32_to_f32

Converts 32-bit signed integer to 32-bit floating point

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, RoundingMode mode` |

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFFF) == 0) {
  return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32(sign, 0x9C, magnitude_of_A, mode);
```

### E.117. ui32_to_f32

Converts 32-bit unsigned integer to 32-bit floating point

| | |
|---|---|
| **Return Type** | `U32` |
| **Arguments** | `U32 a, RoundingMode mode` |

```
if (a == 0) {
  return a;
}
if (a[31] == 1) {
  return softfloat_roundPackToF32(0, 0x9D, a >> 1 | (a & 1), mode);
```

```
} else {
  return softfloat_normRoundPackToF32(0, 0x9C, a, mode);
}
```

### E.118. mode

Returns the current active privilege mode.

| | |
|---|---|
| **Return Type** | `PrivilegeMode` |
| **Arguments** | None |

```
if (!implemented?(ExtensionName::S) && !implemented?(ExtensionName::U) && !implemented?(ExtensionName::H)) {
  return PrivilegeMode::M;
} else {
  return current_mode;
}
```

### E.119. set_mode_no_refresh

Set the current privilege mode to `new_mode`, but don't refresh interrupts

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode new_mode` |

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
}
```

### E.120. set_mode

Set the current privilege mode to `new_mode`

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `PrivilegeMode new_mode` |

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
  refresh_pending_interrupts();
}
```

### E.121. compatible_mode?

Returns true if target_mode is more privileged than actual_mode.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `PrivilegeMode target_mode, PrivilegeMode actual_mode` |

```
if (target_mode == PrivilegeMode::M) {
  return actual_mode == PrivilegeMode::M;
} else if (target_mode == PrivilegeMode::S) {
```

```
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S);
} else if (target_mode == PrivilegeMode::U) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::U);
} else if (target_mode == PrivilegeMode::VS) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS);
} else if (target_mode == PrivilegeMode::VU) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS) ||
(actual_mode == PrivilegeMode::VU);
}
```

## E.122. exception_handling_mode

Returns the target privilege mode that will handle synchronous exception `exception_code`

| | |
|---|---|
| **Return Type** | `PrivilegeMode` |
| **Arguments** | `ExceptionCode exception_code` |

```
if (mode() == PrivilegeMode::M) {
  return PrivilegeMode::M;
} else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::HS) || (mode() == PrivilegeMode::U) {
  if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
    return PrivilegeMode::HS;
  } else {
    return PrivilegeMode::M;
  }
} else {
  assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU, "Unexpected
mode");
  if (($bits(medeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
    if (($bits(hedeleg) & (MXLEN'1 << $bits(exception_code))) != 0) {
      return PrivilegeMode::VS;
    } else {
      return PrivilegeMode::HS;
    }
  } else {
    return PrivilegeMode::M;
  }
}
```

## E.123. creg2reg

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

*Table 15. Registers specified by the three-bit rs1′, rs2′, and rd′ fields of the CIW, CL, CS, CA, and CB formats.*

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

| | |
|---|---|
| **Return Type** | `Bits⑤` |
| **Arguments** | `Bits<3> creg_idx` |

```
return {2'b01, creg_idx};
```

## E.124. unimplemented_csr

Either raises an IllegalInstruction exception or enters unpredictable state, depending on the setting of the TRAP_ON_UNIMPLEMENTED_CSR parameter.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Bits<INSTR_ENC_SIZE> encoding` |

```
if (TRAP_ON_UNIMPLEMENTED_CSR) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else {
  unpredictable("Accessing an unimplmented CSR");
}
```

### E.125. mtval_readonly?

Returns whether or not CSR[mtval] is read-only based on implementation options

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
return !(REPORT_VA_IN_MTVAL_ON_BREAKPOINT || REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_MTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_MTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
```

### E.126. stval_readonly?

Returns whether or not CSR[stval] is read-only based on implementation options

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
if (implemented?(ExtensionName::S)) {
  return !(REPORT_VA_IN_STVAL_ON_BREAKPOINT || REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_STVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_STVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
  return true;
}
```

### E.127. vstval_readonly?

Returns whether or not CSR[vstval] is read-only based on implementation options

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
if (implemented?(ExtensionName::H)) {
  return !(REPORT_VA_IN_VSTVAL_ON_BREAKPOINT || REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_VSTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
```

```
REPORT_CAUSE_IN_VSTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
  return true;
}
```

### E.128. mtval_for

Given an exception code and a **legal** non-zero value for mtval, returns the value to be written in mtval considering implementation options

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `ExceptionCode exception_code, XReg tval` |

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_MTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

### E.129. stval_for

Given an exception code and a **legal** non-zero value for stval, returns the value to be written in stval considering implementation options

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `ExceptionCode exception_code, XReg tval` |

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_STVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
```

```
  return REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

## E.130. vstval_for

Given an exception code and a **legal** non-zero value for vstval, returns the value to be written in vstval considering implementation options

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `ExceptionCode exception_code, XReg tval` |

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_VSTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

## E.131. raise_guest_page_fault

Raise a guest page fault exception.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `MemoryOperation op, XReg gpa, XReg gva, XReg tinst_value, PrivilegeMode from_mode` |

```
ExceptionCode code;
Boolean write_gpa_in_tval;
if (op == MemoryOperation::Read) {
  code = ExceptionCode::LoadGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_LOAD_GUEST_PAGE_FAULT;
} else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
  code = ExceptionCode::StoreAmoGuestPageFault;
```

```
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_STORE_AMO_GUEST_PAGE_FAULT;
} else {
  assert(op == MemoryOperation::Fetch, "unexpected memory operation");
  code = ExceptionCode::InstructionGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_INSTRUCTION_GUEST_PAGE_FAULT;
}
PrivilegeMode handling_mode = exception_handling_mode(code);
if (handling_mode == PrivilegeMode::S) {
  htval.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  htinst.VALUE = tinst_value;
  sepc.PC = $pc;
  if (!stval_readonly?()) {
    stval.VALUE = stval_for(code, gva);
  }
  $pc = {stvec.BASE, 2'b00};
  scause.INT = 1'b0;
  scause.CODE = $bits(code);
  hstatus.GVA = 1;
  hstatus.SPV = 1;
  hstatus.SPVP = $bits(from_mode)[0];
  mstatus.SPP = $bits(from_mode)[0];
} else {
  assert(handling_mode == PrivilegeMode::M, "unexpected privilege mode");
  mtval2.VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  mtinst.VALUE = tinst_value;
  mstatus.MPP = $bits(from_mode)[1:0];
  if (MXLEN == 64) {
    mstatus.MPV = 1;
  } else {
    mstatush.MPV = 1;
  }
}
set_mode(handling_mode);
abort_current_instruction();
```

## E.132. raise

Raise synchronous exception number `exception_code`.

The exception may be imprecise, and will cause execution to enter an unpredictable state, if PRECISE_SYNCHRONOUS_EXCEPTIONS is false.

Otherwise, the exception will be precise.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval` |

```
if (!PRECISE_SYNCHRONOUS_EXCEPTIONS) {
  unpredictable("Imprecise synchronous exception");
} else {
  raise_precise(exception_code, from_mode, tval);
}
```

## E.133. raise_precise

Raise synchronous exception number `exception_code`.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval` |

```
PrivilegeMode handling_mode = exception_handling_mode(exception_code);
if (handling_mode == PrivilegeMode::M) {
  mepc.PC = $pc;
  if (!mtval_readonly?()) {
```

```
    mtval.VALUE = mtval_for(exception_code, tval);
  }
  $pc = {mtvec.BASE, 2'b00};
  mcause.INT = 1'b0;
  mcause.CODE = $bits(exception_code);
  if (misa.H == 1) {
    mtval2.VALUE = 0;
    mtinst.VALUE = 0;
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      if (MXLEN == 32) {
        mstatush.MPV = 1;
      } else {
        mstatus.MPV = 1;
      }
    } else {
      if (MXLEN == 32) {
        mstatush.MPV = 0;
      } else {
        mstatus.MPV = 0;
      }
    }
  }
  mstatus.MPP = $bits(from_mode);
} else if (misa.S == 1 && (handling_mode == PrivilegeMode::S)) {
  sepc.PC = $pc;
  if (!stval_readonly?()) {
    stval.VALUE = stval_for(exception_code, tval);
  }
  $pc = {stvec.BASE, 2'b00};
  scause.INT = 1'b0;
  scause.CODE = $bits(exception_code);
  mstatus.SPP = $bits(from_mode)[0];
  if (misa.H == 1) {
    htval.VALUE = 0;
    htinst.VALUE = 0;
    hstatus.SPV = $bits(from_mode)[2];
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      hstatus.SPV = 1;
      if (exception_code == ExceptionCode::Breakpoint) && (REPORT_VA_IN_STVAL_ON_BREAKPOINT || exception_code ==
ExceptionCode::LoadAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || exception_code ==
ExceptionCode::StoreAmoAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || exception_code ==
ExceptionCode::InstructionAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || exception_code ==
ExceptionCode::LoadAccessFault) && (REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || exception_code ==
ExceptionCode::StoreAmoAccessFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || exception_code ==
ExceptionCode::InstructionAccessFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || exception_code ==
ExceptionCode::LoadPageFault) && (REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || exception_code ==
ExceptionCode::StoreAmoPageFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || exception_code ==
ExceptionCode::InstructionPageFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT) {
        hstatus.GVA = 1;
      } else {
        hstatus.GVA = 0;
      }
      hstatus.SPVP = $bits(from_mode)[0];
    } else {
      hstatus.SPV = 0;
      hstatus.GVA = 0;
    }
  }
} else if (misa.H == 1 && (handling_mode == PrivilegeMode::VS)) {
  vsepc.PC = $pc;
  if (!vstval_readonly?()) {
    vstval.VALUE = vstval_for(exception_code, tval);
  }
  $pc = {vstvec.BASE, 2'b00};
  vscause.INT = 1'b0;
  vscause.CODE = $bits(exception_code);
  vsstatus.SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();
```

## E.134. ialign

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

| | |
|---|---|
| **Return Type** | `Bits⑥` |
| **Arguments** | None |

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

## E.135. jump

Jump to virtual address `target_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg target_addr` |

```
if ((ialign() == 16) && target_addr & 0x1) != 0 {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
} else if ((ialign() == 32) && (target_addr & 0x3) != 0) {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
}
$pc = target_addr;
```

## E.136. jump_halfword

Jump to virtual halfword address `target_hw_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg target_hw_addr` |

```
assert((target_hw_addr & 0x1) == 0x0, "Expected halfword-aligned address in jump_halfword");
if (ialign() != 16) {
  if ((target_hw_addr & 0x3) != 0) {
    raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_hw_addr);
  }
}
$pc = target_hw_addr;
```

## E.137. valid_interrupt_code?

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

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg code` |

```
if (code > 1 `<< $enum_element_size(InterruptCode - 1)) {
```

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

### E.138. valid_exception_code?

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

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg code` |

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

### E.139. xlen

Returns the effective XLEN for the current privilege mode.

| | |
|---|---|
| **Return Type** | `Bits⑧` |
| **Arguments** | None |

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

```
  }
}
```

### E.140. virtual_mode?

Returns True if the current mode is virtual (VS or VU).

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
return (mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU);
```

### E.141. mask_eaddr

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

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `XReg eaddr` |

```
return eaddr;
```

### E.142. pmp_match_64

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

| | |
|---|---|
| **Return Type** | `PmpMatchResult, PmpCfg` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size` |

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 8) * 2;
  Bits<6> shamt = (i % 8) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<64> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_hi = 0;
  Bits<PHYS_ADDR_WIDTH> range_lo = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_lo = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_lo = (pmpaddr_value & ~mask) << 2;
    Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;
    range_hi = pmpaddr_value & ~mask) + len) << 2;   } else if (cfg.A == $bits(PmpCfg_A::NA4 {
    range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
    range_hi = range_lo + 4;
```

```
  }
  if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

## E.143. pmp_match_32

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

| | |
|---|---|
| **Return Type** | `PmpMatchResult, PmpCfg` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size` |

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 4);
  Bits<6> shamt = (i % 4) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<32> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_hi = 0;
  Bits<PHYS_ADDR_WIDTH> range_lo = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_lo = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_lo = (csr_sw_read(tor_pmpaddr_csr) << 2)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_hi = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 2> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 2> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_lo = (pmpaddr_value & ~mask) << 2;
    Bits<PHYS_ADDR_WIDTH - 2> len = mask + 1;
    range_hi = pmpaddr_value & ~mask) + len) << 2;   } else if (cfg.A == $bits(PmpCfg_A::NA4 {
    range_lo = (pmpaddr_csr_value << 2)[PHYS_ADDR_WIDTH - 1:0];
    range_hi = range_lo + 4;
  }
  if ((paddr >= range_lo) && paddr + (access_size / 8 < range_hi)) {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

## E.144. pmp_match

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

| | |
|---|---|
| **Return Type** | `PmpMatchResult, PmpCfg` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size` |

```
if (MXLEN == 64) {
  return pmp_match_64(paddr, access_size);
} else {
  return pmp_match_32(paddr, access_size);
}
```

### E.145. mpv

Returns the current value of CSR[mstatus].MPV (when MXLEN == 64) of CSR[mstatush].MPV (when MXLEN == 32)

| | |
|---|---|
| **Return Type** | `Bits①` |
| **Arguments** | None |

```
if (implemented?(ExtensionName::H)) {
  return (MXLEN == 32) ? mstatush.MPV : mstatus.MPV;
} else {
  assert(false, "TODO");
}
```

### E.146. effective_ldst_mode

Returns the effective privilege mode for normal explicit loads and stores, taking into account the current actual privilege mode and modifications from mstatus.MPRV.

| | |
|---|---|
| **Return Type** | `PrivilegeMode` |
| **Arguments** | None |

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

### E.147. pmp_check

Given a physical address and operation type, return whether or not the access is allowed by PMP.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, MemoryOperation type` |

```
PrivilegeMode mode = effective_ldst_mode();
PmpMatchResult match_result;
PmpCfg cfg;
(match_result, cfg = pmp_match(paddr, access_size));
if (match_result == PmpMatchResult::FullMatch) {
  if (mode == PrivilegeMode::M && (cfg.L == 0)) {
```

```
    return true;
  }
  if (type == MemoryOperation::Write && (cfg.W == 0)) {
    return false;
  } else if (type == MemoryOperation::Read && (cfg.R == 0)) {
    return false;
  } else if (type == MemoryOperation::Fetch && (cfg.X == 0)) {
    return false;
  }
} else if (match_result == PmpMatchResult::NoMatch) {
  if (mode == PrivilegeMode::M) {
    return true;
  } else {
    return false;
  }
} else {
  assert(match_result == PmpMatchResult::PartialMatch, "PMP matching logic error");
  return false;
}
return true;
```

## E.148. access_check

Checks if the physical address paddr is able to access memory, and raises the appropriate exception if not.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, XReg vaddr, MemoryOperation type, ExceptionCode fault_type, PrivilegeMode from_mode` |

```
if (paddr > 1 << PHYS_ADDR_WIDTH) - access_size {
  raise(fault_type, from_mode, vaddr);
}
if (implemented?(ExtensionName::Smpmp)) {
  if (!pmp_check(paddr[PHYS_ADDR_WIDTH - 1:0], access_size, type)) {
    raise(fault_type, from_mode, vaddr);
  }
}
```

## E.149. base32?

return True iff current effective XLEN == 32

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
if (MXLEN == 32) {
  return true;
} else {
  XRegWidth xlen32 = XRegWidth::XLEN32;
  if (mode() == PrivilegeMode::M) {
    return misa.MXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    return mstatus.SXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    return mstatus.UXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    return hstatus.VSXL == $bits(xlen32);
  } else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU, "Unexpected mode");
    return vsstatus.UXL == $bits(xlen32);
  }
}
```

## E.150. base64?

return True iff current effective XLEN == 64

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

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

## E.151. current_translation_mode

Returns the current first-stage translation mode for an explicit load or store from mode given the machine state (e.g., value of satp or vsatp csr).

Returns SatpMode::Reserved if the setting found in satp or vsatp is invalid.

| | |
|---|---|
| **Return Type** | `SatpMode` |
| **Arguments** | `PrivilegeMode mode` |

```
PrivilegeMode effective_mode = effective_ldst_mode();
if (effective_mode == PrivilegeMode::M) {
  return SatpMode::Bare;
}
if (misa.H == 1'b1) {
  if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU) {
    Bits<4> mode_val = vsatp.MODE;
    if (mode_val == $bits(SatpMode::Sv32)) {
      if (MXLEN == 64) {
        if ((effective_mode == PrivilegeMode::VS) && (hstatus.VSXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
        if ((effective_mode == PrivilegeMode::VU) && (vsstatus.UXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
      }
      if (!SV32_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv32;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV39_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv39;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV48_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv48;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
      if (effective_mode == PrivilegeMode::VS && hstatus.VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && vsstatus.UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
```

```
      }
      if (!SV57_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv57;
    } else {
      return SatpMode::Reserved;
    }
  }
} else if (misa.S == 1'b1) {
  assert(effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U, "unexpected priv mode");
  Bits<4> mode_val = satp.MODE;
  if (mode_val == $bits(SatpMode::Sv32)) {
    if (MXLEN == 64) {
      if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
    }
    if (!implemented?(ExtensionName::Sv32)) {
      return SatpMode::Reserved;
    }
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv39)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv39;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv48)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv48;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
    if (effective_mode == PrivilegeMode::S && mstatus.SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && sstatus.UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv57)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv57;
  } else {
    return SatpMode::Reserved;
  }
}
```

## E.152. current_gstage_translation_mode

Returns the current second-stage translation mode for a load or store from VS-mode or VU-mode.

| | |
|---|---|
| **Return Type** | `HgatpMode` |
| **Arguments** | None |

```
return $enum(HgatpMode, hgatp.MODE);
```

### E.153. translate_gstage

Translates a guest physical address to a physical address.

| | |
|---|---|
| **Return Type** | `TranslationResult` |
| **Arguments** | `XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding` |

```
TranslationResult result;
if (effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U) {
  result.paddr = gpaddr;
  return result;
}
Boolean mxr = mstatus.MXR == 1;
if (GSTAGE_MODE_BARE && hgatp.MODE == $bits(HgatpMode::Bare)) {
  result.paddr = gpaddr;
  return result;
} else if (SV32X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv32x4)) {
  return gstage_page_walk<32, 34, 32, 2>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV39X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv39x4)) {
  return gstage_page_walk<39, 56, 64, 3>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV48X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv48x4)) {
  return gstage_page_walk<48, 56, 64, 4>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV57X4_TRANSLATION && hgatp.MODE == $bits(HgatpMode::Sv57x4)) {
  return gstage_page_walk<57, 56, 64, 5>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else {
  if (op == MemoryOperation::Read) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else {
    assert(op == MemoryOperation::Fetch, "unexpected memory op");
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  }
}
```

### E.154. tinst_value_for_guest_page_fault

Returns the value of htinst/mtinst for a Guest Page Fault

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `MemoryOperation op, Bits<INSTR_ENC_SIZE> encoding, Boolean for_final_vs_pte` |

```
if (for_final_vs_pte) {
  if (op == MemoryOperation::Fetch) {
    if (TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else {
      assert(TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always pseudoinstruction", "Instruction guest page
faults can only report zero/pseudo instruction in tval");
      return 0x00002000;
    }
  } else if (op == MemoryOperation::Read) {
    if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002000;
      } else {
        return 0x00003000;
      }
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always transformed standard instruction") {
```

```
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002020;
      } else {
        return 0x00003020;
      }
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  }
} else {
  if (REPORT_GPA_IN_TVAL_ON_INTERMEDIATE_GUEST_PAGE_FAULT) {
    if ((VSXLEN == 32) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN32)) {
      return 0x00002000;
    } else if ((VSXLEN == 64) || MXLEN == 64) && (hstatus.VSXL == $bits(XRegWidth::XLEN64)) {
      return 0x00003000;
    }
  }
}
```

## E.155. tinst_transform

Returns the standard transformation of an encoding for htinst/mtinst

| | |
|---|---|
| **Return Type** | `Bits<INSTR_ENC_SIZE>` |
| **Arguments** | `Bits<INSTR_ENC_SIZE> encoding, Bits<5> addr_offset` |

```
if (encoding[1:0] == 0b11) {
  if (encoding[6:2] == 5'b00001) {
    return {{12{1'b0}}, addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b01000) {
    return {{7{1'b0}}, encoding[24:20], addr_offset, encoding[14:12], {5{1'b0}}, encoding[6:0]};
  } else if (encoding[6:2] == 5'b01011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b00011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else {
    assert(false, "Bad transform");
  }
} else {
  assert(false, "TODO: compressed instruction");
}
```

## E.156. transformed_standard_instruction_for_tinst

Transforms an instruction encoding for htinst.

| | |
|---|---|
| **Return Type** | `Bits<INSTR_ENC_SIZE>` |
| **Arguments** | `Bits<INSTR_ENC_SIZE> original` |

```
assert(false, "TODO");
return 0;
```

## E.157. tinst_value

Returns the value of htinst/mtinst for the given exception code.

| | |
|---|---|
| **Return Type** | `XReg` |
| **Arguments** | `ExceptionCode code, Bits<INSTR_ENC_SIZE> encoding` |

```
if (code == ExceptionCode::InstructionAddressMisaligned) {
  if (TINST_VALUE_ON_INSTRUCTION_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to an InstructionAddressMisaligned
exception");
  }
} else if (code == ExceptionCode::InstructionAccessFault) {
  return 0;
} else if (code == ExceptionCode::IllegalInstruction) {
  return 0;
} else if (code == ExceptionCode::Breakpoint) {
  if (TINST_VALUE_ON_BREAKPOINT == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a Breakpoint exception");
  }
} else if (code == ExceptionCode::VirtualInstruction) {
  if (TINST_VALUE_ON_VIRTUAL_INSTRUCTION == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VirtualInstruction exception");
  }
} else if (code == ExceptionCode::LoadAddressMisaligned) {
  if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadAddressMisaligned exception");
  }
} else if (code == ExceptionCode::LoadAccessFault) {
  if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadAccessFault exception");
  }
} else if (code == ExceptionCode::StoreAmoAddressMisaligned) {
  if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAddressMisaligned exception");
  }
} else if (code == ExceptionCode::StoreAmoAccessFault) {
  if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAccessFault exception");
  }
} else if (code == ExceptionCode::Ucall) {
  if (TINST_VALUE_ON_UCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a UCall exception");
  }
} else if (code == ExceptionCode::Scall) {
  if (TINST_VALUE_ON_SCALL == "always zero") {
```

```
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a SCall exception");
  }
} else if (code == ExceptionCode::Mcall) {
  if (TINST_VALUE_ON_MCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a MCall exception");
  }
} else if (code == ExceptionCode::VScall) {
  if (TINST_VALUE_ON_VSCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VSCall exception");
  }
} else if (code == ExceptionCode::InstructionPageFault) {
  return 0;
} else if (code == ExceptionCode::LoadPageFault) {
  if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadPageFault exception");
  }
} else if (code == ExceptionCode::StoreAmoPageFault) {
  if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoPageFault exception");
  }
} else {
  assert(false, "Unhandled exception type");
}
```

### E.158. gstage_page_walk

Translate guest physical address to physical address through a page walk.

May raise a Guest Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated physical address.

| | |
|---|---|
| **Return Type** | `TranslationResult` |
| **Arguments** | `XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Boolean for_final_vs_pte, Bits<INSTR_ENC_SIZE> encoding` |

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadGuestPageFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionGuestPageFault : ExceptionCode::StoreAmoGuestPageFault);
Boolean mxr = for_final_vs_pte && (mstatus.MXR == 1);
Boolean pbmte = menvcfg.PBMTE == 1;
Boolean adue = menvcfg.ADUE == 1;
Bits<32> tinst = tinst_value_for_guest_page_fault(op, encoding, for_final_vs_pte);
U32 max_gpa_width = LEVELS * VPN_SIZE + 2 + 12;
if (gpaddr >> max_gpa_width != 0) {
  raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
```

```
}
ppn = hgatp.PPN;
for (U32 i = (LEVELS - 1); i >= 0; i--) {
  U32 this_vpn_size = (i == (LEVELS - 1)) ? VPN_SIZE + 2 : VPN_SIZE;
  U32 vpn = (gpaddr >> (12 + VPN_SIZE * i)) & 1 << this_vpn_size) - 1);   Bits<PA_SIZE> pte_paddr = (ppn << 12) + (vpn
* (PTESIZE / 8;
  if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_paddr, PTESIZE)) {
    raise(access_fault_code, PrivilegeMode::U, vaddr);
  }
  access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
  Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_paddr);
  PteFlags pte_flags = pte[9:0];
  if ((VA_SIZE != 32) && (pte[60:54] != 0)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (!implemented?(ExtensionName::Svnapot)) {
    if ((PTESIZE >= 64) && pte[63] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
  }
  if ((PTESIZE >= 64) && !pbmte && (pte[62:61] != 0)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if ((PTESIZE >= 64) && pbmte && (pte[62:61] == 3)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.V == 0) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.R == 0 && pte_flags.W == 1) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.R == 1 || pte_flags.X == 1) {
    if (pte_flags.U == 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    } else if ((op == MemoryOperation::Read) || (op == MemoryOperation::ReadModifyWrite)) {
      if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
      }
    }
    if ((i > 0) && (pte[(i - 1) * VPN_SIZE:10] != 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op ==
MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_paddr, PTESIZE)) {
          raise(access_fault_code, PrivilegeMode::U, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_paddr, PTESIZE)) {
          raise(access_fault_code, PrivilegeMode::U, vaddr);
        }
        access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
          success = atomic_check_then_write_64(pte_paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
          i = i + 1;
        } else {
```

```
          result.paddr = pte_paddr;
          if (PTESIZE >= 64) {
            result.pbmt = $enum(Pbmt, pte[62:61]);
          }
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
      }
    }
  } else {
    if (i == 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
    i = i - 1;
  }
}
```

### E.159. stage1_page_walk

Translate virtual address to physical address through a page walk.

May raise a Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated guest physical address.

| | |
|---|---|
| **Return Type** | `TranslationResult` |
| **Arguments** | `Bits<MXLEN> vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding` |

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadPageFault : (op ==
MemoryOperation::Fetch ? ExceptionCode::InstructionPageFault : ExceptionCode::StoreAmoPageFault);
Boolean sse = false;
Boolean adue;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  adue = henvcfg.ADUE == 1;
} else {
  adue = menvcfg.ADUE == 1;
}
Boolean pbmte;
if (VA_SIZE == 32) {
  pbmte = false;
} else {
  if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    pbmte = henvcfg.PBMTE == 1;
  } else {
    pbmte = menvcfg.PBMTE == 1;
  }
}
```

```
Boolean mxr;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  mxr = (mstatus.MXR == 1) || (vsstatus.MXR == 1);
} else {
  mxr = mstatus.MXR == 1;
}
Boolean sum;
if (misa.H == 1 && (effective_mode == PrivilegeMode::VS)) {
  sum = vsstatus.SUM == 1;
} else {
  sum = mstatus.SUM == 1;
}
ppn = vsatp.PPN;
if ((VA_SIZE < xlen()) && (vaddr[xlen() - 1:VA_SIZE] != {xlen() - VA_SIZE{vaddr[VA_SIZE - 1]}})) {
  raise(page_fault_code, effective_mode, vaddr);
}
for (U32 i = (LEVELS - 1); i >= 0; i--) {
  U32 vpn = (vaddr >> (12 + VPN_SIZE * i)) & 1 `<< VPN_SIZE) - 1);   Bits<PA_SIZE> pte_gpaddr = (ppn << 12) + (vpn *
(PTESIZE / 8;
  TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Read, effective_mode, encoding);
  if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_phys.paddr, PTESIZE)) {
    raise(access_fault_code, effective_mode, vaddr);
  }
  access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
  Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_phys.paddr);
  PteFlags pte_flags = pte[9:0];
  Boolean ss_page = (pte_flags.R == 0) && (pte_flags.W == 1) && (pte_flags.X == 0);
  if ((VA_SIZE != 32) && (pte[60:54] != 0)) {
    raise(page_fault_code, effective_mode, vaddr);
  }
  if (pte_flags.V == 0) {
    raise(page_fault_code, effective_mode, vaddr);
  }
  if (!sse) {
    if ((pte_flags.R == 0) && (pte_flags.W == 1)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  }
  if (pbmte) {
    if (pte[62:61] == 3) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  } else {
    if ((PTESIZE >= 64) && (pte[62:61] != 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  }
  if (!implemented?(ExtensionName::Svnapot)) {
    if ((PTESIZE >= 64) && (pte[63] != 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
  }
  if (pte_flags.R == 1 || pte_flags.X == 1) {
    if (op == MemoryOperation::Read || op == MemoryOperation::ReadModifyWrite) {
      if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
        raise(page_fault_code, effective_mode, vaddr);
      }
      if (effective_mode == PrivilegeMode::U && pte_flags.U == 0) {
        raise(page_fault_code, effective_mode, vaddr);
      } else if (misa.H == 1 && effective_mode == PrivilegeMode::VU && pte_flags.U == 0) {
        raise(page_fault_code, effective_mode, vaddr);
      } else if (effective_mode == PrivilegeMode::S && pte_flags.U == 1 && !sum) {
        raise(page_fault_code, effective_mode, vaddr);
      } else if (effective_mode == PrivilegeMode::VS && pte_flags.U == 1 && !sum) {
        raise(page_fault_code, effective_mode, vaddr);
      }
    }
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    } else if ((op == MemoryOperation::Fetch) && ss_page) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    raise(page_fault_code, effective_mode, vaddr) if;
```

```
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op ==
MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Write, effective_mode,
encoding);
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_phys.paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
          success = atomic_check_then_write_64(pte_phys.paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
          i = i + 1;
        } else {
          TranslationResult pte_phys = translate_gstage({(pte[PA_SIZE - 3:(i * VPN_SIZE) + 10] << 2), vaddr[11:0]},
vaddr, op, effective_mode, encoding);
          result.paddr = pte_phys.paddr;
          result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
        raise(page_fault_code, effective_mode, vaddr);
      }
    }
    TranslationResult pte_phys = translate_gstage({(pte[PA_SIZE - 3:(i * VPN_SIZE) + 10] << 2), vaddr[11:0]}, vaddr, op,
effective_mode, encoding);
    result.paddr = pte_phys.paddr;
    if (PTESIZE >= 64) {
      result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
    }
    result.pte_flags = pte_flags;
    return result;
  } else {
    if (i == 0) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
      raise(page_fault_code, effective_mode, vaddr);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
  }
}
```

## E.160. translate

Translate a virtual address for operation type op that appears to execute at effective_mode.

The translation will depend on the effective privilege mode.

May raise a Page Fault or Access Fault.

The final physical address is **not** access checked (for PMP, PMA, etc., violations). (though intermediate page table reads will be)

| | |
|---|---|
| **Return Type** | `TranslationResult` |
| **Arguments** | `XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding` |

```
Boolean cached_translation_valid;
CachedTranslationResult cached_translation_result;
cached_translation_result = cached_translation(vaddr, op);
if (cached_translation_result.valid) {
  return cached_translation_result.result;
}
TranslationResult result;
if (effective_mode == PrivilegeMode::M) {
  result.paddr = vaddr;
  return result;
}
SatpMode translation_mode = current_translation_mode(effective_mode);
if (translation_mode == SatpMode::Reserved) {
  if (op == MemoryOperation::Read) {
    raise(ExceptionCode::LoadPageFault, effective_mode, vaddr);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise(ExceptionCode::StoreAmoPageFault, effective_mode, vaddr);
  } else {
    assert(op == MemoryOperation::Fetch, "Unexpected memory operation");
    raise(ExceptionCode::InstructionPageFault, effective_mode, vaddr);
  }
}
if (translation_mode == SatpMode::Sv32) {
  result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv39) {
  result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv48) {
  result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode, encoding);
} else if (translation_mode == SatpMode::Sv57) {
  result = stage1_page_walk<57, 56, 64, 5>(vaddr, op, effective_mode, encoding);
} else {
  assert(false, "Unexpected SatpMode");
}
maybe_cache_translation(vaddr, op, result);
return result;
```

### E.161. canonical_vaddr?

Returns whether or not *vaddr* is a valid (*i.e.*, canonical) virtual address.

If pointer masking (S**pm) is enabled, then vaddr will be masked before checking the canonical address.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg vaddr` |

```
if (misa.S == 1'b0) {
  return true;
}
SatpMode satp_mode;
if (virtual_mode?()) {
  satp_mode = $enum(SatpMode, vsatp.MODE);
} else {
  satp_mode = $enum(SatpMode, satp.MODE);
}
XReg eaddr = mask_eaddr(vaddr);
if (satp_mode == SatpMode::Bare) {
  return true;
} else if (satp_mode == SatpMode::Sv32) {
  return true;
} else if (satp_mode == SatpMode::Sv39) {
```

```
  return eaddr[63:39] == {25{eaddr[38]}};
} else if (satp_mode == SatpMode::Sv48) {
  return eaddr[63:48] == {16{eaddr[47]}};
} else if (satp_mode == SatpMode::Sv57) {
  return eaddr[63:57] == {6{eaddr[56]}};
}
```

### E.162. canonical_gpaddr?

Returns whether or not gpaddr is a valid (*i.e.*, canonical) guest physical address.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `XReg gpaddr` |

```
SatpMode satp_mode = $enum(SatpMode, satp.MODE);
if (satp_mode == SatpMode::Bare) {
  return true;
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv39)) {
  return gpaddr[63:39] == {25{gpaddr[38]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv48)) {
  return gpaddr[63:48] == {16{gpaddr[47]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv57)) {
  return gpaddr[63:57] == {6{gpaddr[56]}};
}
```

### E.163. misaligned_is_atomic?

Returns true if an access starting at physical_address that is N bits long is atomic.

This function takes into account any Atomicity Granule PMAs, so **it should not be used for load-reserved/store-conditional**, since those PMAs do not apply to those accesses.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<PHYS_ADDR_WIDTH> physical_address` |

```
return false if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE == 0);
if (pma_applies?(PmaAttribute::MAG16, physical_address, N) && in_naturally_aligned_region?<128>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG8, physical_address, N) && in_naturally_aligned_region?<4>(physical_address,
N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG4, physical_address, N) && in_naturally_aligned_region?<32>(physical_address,
N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG2, physical_address, N) && in_naturally_aligned_region?<16>(physical_address,
N)) {
  return true;
} else {
  return false;
}
```

### E.164. read_memory_aligned

Read from virtual memory using a known aligned address.

| | |
|---|---|
| **Return Type** | `Bits<LEN>` |

| Arguments | XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding |
|---|---|

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
return read_physical_memory<LEN>(result.paddr);
```

## E.165. read_memory

Read from virtual memory.

| Return Type | Bits<LEN> |
|---|---|
| Arguments | XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding |

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  return read_memory_aligned<LEN>(virtual_address, encoding);
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with
large atomicity granule");
  physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
    return read_physical_memory<LEN>(physical_address);
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::LoadAddressMisaligned, effective_ldst_mode(), virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    Bits<LEN> result = 0;
    for (U32 i = 0; i <= (LEN / 8); i++) {
      result = result | (read_memory_aligned<8>(virtual_address + i, encoding) << (8 * i));
    }
    return result;
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior
when any part of the misaligned access causes an exception");
  }
}
```

## E.166. read_memory_xlen

Read XLEN bits from memory

| Return Type | Bits<MXLEN> |
|---|---|

| | |
|---|---|
| **Arguments** | `XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding` |

```
if (xlen() == 32) {
  return read_memory<32>(virtual_address, encoding);
} else {
  return read_memory<64>(virtual_address, encoding);
}
```

### E.167. write_memory_xlen

Read XLEN bits from memory

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding` |

```
if (xlen() == 32) {
  return write_memory<32>(virtual_address, value, encoding);
} else {
  return write_memory<64>(virtual_address, value, encoding);
}
```

### E.168. read_memory_xlen_aligned

Read from virtual memory XLEN (which may be runtime-determined) bits using a known aligned address.

| | |
|---|---|
| **Return Type** | `Bits<MXLEN>` |
| **Arguments** | `XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding` |

```
TranslationResult result;
if (misa.S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
  return read_physical_memory<32>(result.paddr);
} else {
  return read_physical_memory<64>(result.paddr);
}
```

### E.169. invalidate_reservation_set

Invalidates any currently held reservation set.

*This function may be called by the platform, independent of any actions occurring in the local hart, for any or no reason.*

*The platform* **must** *call this function if an external hart or device accesses part of this reservation set while reservation_set_valid could be true.*

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | None |

```
reservation_set_valid = false;
```

### E.170. register_reservation_set

Register a reservation for a physical address range that subsumes [physical_address, physical_address + N).

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `Bits<MXLEN> physical_address, Bits<MXLEN> length` |

```
reservation_set_valid = true;
reservation_set_address = physical_address;
if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 64-byte region") {
  reservation_set_address = physical_address & ~MXLEN'h3f;
  reservation_set_size = 64;
} else if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 128-byte region") {
  reservation_set_address = physical_address & ~MXLEN'h7f;
  reservation_set_size = 128;
} else if (LRSC_RESERVATION_STRATEGY == "reserve exactly enough to cover the access") {
  reservation_set_address = physical_address;
  reservation_set_size = length;
} else if (LRSC_RESERVATION_STRATEGY == "custom") {
  unpredictable("Implementations may set reservation sets of any size, as long as they cover the reserved accessed");
} else {
  assert(false, "Unexpected LRSC_RESERVATION_STRATEGY");
}
```

### E.171. load_reserved

Register a reservation for virtual_address at least N bits long and read the value from memory.

If aq is set, then also perform a memory model acquire.

If rl is set, then also perform a memory model release (software is discouraged from doing so).

This function assumes alignment checks have already occurred.

| | |
|---|---|
| **Return Type** | `Bits<N>` |
| **Arguments** | `Bits<MXLEN> virtual_address, Bits<1>  aq, Bits<1>  rl, Bits<INSTR_ENC_SIZE> encoding` |

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Read,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::LoadAccessFault, effective_ldst_mode(), virtual_address);
}
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
register_reservation_set(physical_address, N);
if (misa.S == 1 && LRSC_FAIL_ON_VA_SYNONYM) {
  reservation_virtual_address = virtual_address;
}
return read_memory_aligned<N>(physical_address, encoding);
```

### E.172. store_conditional

Atomically check the reservation set to ensure:

- it is valid
- it covers the region addressed by this store
- the address setting the reservation set matches virtual address

If the preceding are met, perform the store and return 0. Otherwise, return 1.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | `Bits<MXLEN> virtual_address, Bits<MXLEN> value, Bits<1>  aq, Bits<1>  rl, Bits<INSTR_ENC_SIZE> encoding` |

```
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
}
access_check(physical_address, N, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
if (reservation_set_valid == false) {
  return false;
}
if (!contains?(reservation_set_address, reservation_set_size, physical_address, N)) {
  invalidate_reservation_set();
  return false;
}
if (LRSC_FAIL_ON_NON_EXACT_LRSC) {
  if (reservation_physical_address != physical_address || reservation_size != N) {
    invalidate_reservation_set();
    return false;
  }
}
if (LRSC_FAIL_ON_VA_SYNONYM) {
  if (reservation_virtual_address != virtual_address || reservation_size != N) {
    invalidate_reservation_set();
    return false;
  }
}
write_physical_memory<N>(physical_address, value);
return true;
```

## E.173. amo

Atomically read-modify-write the location at virtual_address.

The value written to virtual_address will depend on op.

If aq is 1, then the amo also acts as a memory model acquire. If rl is 1, then the amo also acts as a memory model release.

| | |
|---|---|
| **Return Type** | `Bits<N>` |
| **Arguments** | `XReg virtual_address, Bits<N> value, AmoOperation op, Bits<1>  aq, Bits<1>  rl, Bits<INSTR_ENC_SIZE> encoding` |

```
Boolean aligned = is_naturally_aligned<N>(virtual_address);
if (!aligned && MISALIGNED_LDST_EXCEPTION_PRIORITY == "high") {
  raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
}
Bits<PHYS_ADDR_WIDTH> physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::ReadModifyWrite,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::AmoNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
} else if (op == AmoOperation::Add || op == AmoOperation::Max || op == AmoOperation::Maxu || op == AmoOperation::Min ||
op == AmoOperation::Minu && !pma_applies?(PmaAttribute::AmoArithmetic, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
} else if (op == AmoOperation::And || op == AmoOperation::Or || op == AmoOperation::Xor && !
```

```
pma_applies?(PmaAttribute::AmoLogical, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, effective_ldst_mode(), virtual_address);
} else {
  assert(pma_applies?(PmaAttribute::AmoSwap, physical_address, N) && op == AmoOperation::Swap, "Bad AMO operation");
}
if (!aligned && !misaligned_is_atomic?<N>(physical_address)) {
  raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
}
if (N == 32) {
  return atomic_read_modify_write_32(physical_address, value, op);
} else {
  return atomic_read_modify_write_64(physical_address, value, op);
}
```

## E.174. write_memory_aligned

Write to virtual memory using a known aligned address.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding` |

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
write_physical_memory<LEN>(physical_address, value);
```

## E.175. write_memory

Write to virtual memory

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding` |

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  write_memory_aligned<LEN>(virtual_address, value, encoding);
  return ;
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with
large atomicity granule");
  physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
    write_physical_memory<LEN>(physical_address, value);
    return ;
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::StoreAmoAddressMisaligned, effective_ldst_mode(), virtual_address);
} else {
```

```
  if (MISALIGNED_SPLIT_STRATEGY == "by_byte") {
    for (U32 i = 0; i <= (LEN / 8); i++) {
      write_memory_aligned<8>(virtual_address + i, (value >> (8 * i))[7:0], encoding);
    }
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior
when any part of the misaligned access causes an exception");
  }
}
```

## E.176. write_memory_xlen_aligned

Write to virtual memory XLEN bits (which may be runtime determined) using a known aligned address.

| | |
|---|---|
| **Return Type** | `void` |
| **Arguments** | `XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding` |

```
XReg physical_address;
physical_address = (misa.S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
access_check(physical_address, xlen(), virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
  write_physical_memory<32>(physical_address, value);
} else {
  write_physical_memory<64>(physical_address, value);
}
```

## E.177. mstatus_sd_has_known_reset

Returns true if the mstatus.SD bit has a defined reset value, as determined by various parameters.

| | |
|---|---|
| **Return Type** | `Boolean` |
| **Arguments** | None |

```
Boolean fs_has_single_value = !implemented?(ExtensionName::F || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1));
Boolean vs_has_single_value = !implemented?(ExtensionName::V || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1));
return fs_has_single_value && vs_has_single_value;
```

## E.178. mstatus_sd_reset_value

Returns the reset value of mstatus.SD when known

| | |
|---|---|
| **Return Type** | `Bits①` |
| **Arguments** | None |

```
assert(mstatus_sd_has_known_reset(), "mstatus_sd_reset_value is only defined when mstatus_sd_has_known_reset() ==
true");
Bits<2> fs_value, vs_value;
if ((!implemented?(ExtensionName::F)) || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1)) {
  fs_value = (!implemented?(ExtensionName::F)) ? 0 : MSTATUS_FS_LEGAL_VALUES[0];
}
if ((!implemented?(ExtensionName::V)) || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1)) {
  fs_value = (!implemented?(ExtensionName::V)) ? 0 : MSTATUS_VS_LEGAL_VALUES[0];
}
return fs_value == 3) || (vs_value == 3 ? 1 : 0;
```