RVA22 Profile Release

Table of Contents

Copyright and license information . . . 1
Acknowledgements . . . 1
1. RISC-V Profiles . . . 1
2. RVA Profile Family . . . 3
3. RVA22 Profile Release . . . 5
Appendix A: Profile Comparisons . . . 17
Appendix B: Extension Details . . . 22
Appendix C: Instruction Details . . . 66
Appendix D: CSR Details . . . 344
Appendix E: IDL Function Details . . . 495

TODO: revmark

Copyright and license information

This document is released under the Creative Commons Attribution 4.0 International License.

Copyright 2023 by RISC-V International.

Acknowledgements

Contributors to the RVA22 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. RVA Profile Family

The RVA profile family targets application processors for markets requiring a high-degree of binary compatibility between compliant implementations.

2.1. RVA Description

RISC-V was designed to provide a highly modular and extensible instruction set and includes a large and growing set of standard extensions, where each standard extension is a bundle of instruction-set features. This is no different than other industry ISAs that continue to add new ISA features. Unlike other ISAs, however, RISC-V has a broad set of contributors and implementers, and also allows users to add their own custom extensions. For some deep embedded markets, highly customized processor configurations are desirable for efficiency, and all software is compiled, ported, and/or developed in-house by the same organization for that specific processor configuration. However, for other markets that expect a substantial fraction of software to be delivered to end-customers in binary form, compatibility across multiple implementations from different RISC-V vendors is required.

The RVIA ISA extension ratification process ensures that all processor vendors have agreed to the specification of a standard extension if present. However, by themselves, the ISA extension specifications do not guarantee that a certain set of standard extensions will be present in all implementations.

The primary goal of the RVA profiles is to align processor vendors targeting binary software markets, so software can rely on the existence of a certain set of ISA features in a particular generation of RISC-V implementations.

Alignment is not only for compatibility, but also to ensure RISC-V is competitive in these markets. The binary app markets are also generally those with the most competitive performance requirements (e.g., mobile, client, server). RVIA cannot mandate the ISA features that a RISC-V binary software ecosystem should use, as each ecosystem will typically select the lowest-common denominator they empirically observe in the deployed devices in their target markets. But RVIA can align hardware vendors to support a common set of features in each generation through the RVA profiles. Without proactive alignment through RVA profiles, RISC-V will be uncompetitive, as even if a particular vendor implements a certain feature, if other vendors do not, then binary distributions will not generally use that feature and all implementations will suffer. While certain features may be discoverable, and alternate code provided in case of presence/absence of a feature, the added cost to support such options is only justified for certain limited cases, and binary app markets will not support a wide range of optional features, particularly for the nascent RISC-V binary app ecosystems.

To maintain alignment and increase RISC-V competitiveness over time, the mandatory set of extensions must increase over time in successive generations of RVA profile. (RVA profiles may eventually have to deprecate previously mandatory instructions, but that is unlikely in the near future.) Note that the RISC-V ISA will continue to evolve, regardless of whether a given software ecosystem settles on a certain generation of profile as the baseline for their ecosystem for many years or even decades. There are many existing binary software ecosystems, which will migrate to RISC-V and evolve at different rates, and more new ones will doubtless be created over the hopefully long lifetime of RISC-V. High-performance application processors require considerable investment, and no single binary app ecosystem can justify the development costs of these processors, especially for RISC-V in its early stage of adoption.

While the heart of the profile is the set of mandatory extensions, there are several kinds of optional extension that serve important roles in the profile.

The first kind are *localized options*, whose presence or use necessarily differs along geo-political and/or jurisdictional boundaries, with crypto being the obvious example. These will always be optional. At least for crypto, discovery has been found to be perfectly acceptable to handle this optionality on other architectures, as the use of the extensions is well contained in certain libraries.

The second kind of optional extension is a *development option*, which represents a new ISA extension in an early part of its lifecycle but which is intended to become mandatory in a later generation of the RVA profile. Processor vendors and software toolchain providers will have varying development schedules, and providing an optional phase in a new extension's lifecycle provides some flexibility while maintaining overall alignment, and is particularly appropriate when hardware or software development for the extension is complex. Denoting an extension as a *development option* signals to the community that development should be prioritized for such extensions as they will become mandatory.

The third kind of optional extension are *expansion options*, which are those that may have a large implementation cost but are not always needed in a particular platform, and which can be readily handled by discovery. These are also intended to remain available as expansion options in future versions of the profile. Several supervisor-mode extensions fall into this category, e.g., Sv57, which has a notable PPA impact over Sv48 and is not needed on smaller platforms. Some unprivileged extensions that may fall into this category are possible future matrix extensions. These have large implementation costs, and use of matrix instructions can be readily supported with discovery and alternate math libraries.

The fourth kind of optional extensions are *transitory options*, where it is not clear if the extension will change to a mandatory, localized, or expansion option, or be possibly dropped over time. Cryptography provides some examples where earlier cyphers have been broken and are now deprecated. RVIA used this mechanism to enable scalar crypto until vector crypto was ready. Software security features may also be in this category, with examples of deprecated security features occurring in other architectures. As another example, the recent avalanche of new numeric datatypes for AI/ML may eventually subside with a few survivors actually being used longer term. Denoting an option as transitory signals to the community that this extension may be removed in a future profile, though the time scale may span many years.

Except for the localized options, it could be argued that other three kinds of option could be left out of profiles. Binary distributions of applications willing to invest in discovery can use an optional extension, and customers compiling their own applications can take advantage of the feature on a particular implementation, even when that system is mostly running binary distributions that ignore the new extension. However, there is value in providing guidance to align hardware vendors and software developers around what extensions are worth implementing and worth discovering, by designating only a few important features as profile options and limiting their granularity.

2.2. RVA Naming Scheme

The profile family name is RVA (RISC-V Apps processor). 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 **RVA** for RISC-V Applications
- Profile release
- Privilege mode:
  - **U** Unprivileged (available to any privilege mode, **U** is **not** User-mode)
  - **S** Supervisor mode (note that Hypervisor support is treated as an option)
  - **M** Machine 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. RVA Profile Releases

The following profile releases are defined in this profile family:

Name
: RVA20

State
: ratified

Ratification date

2023-04-03

Name

RVA22

State

ratified

Ratification date

2023-04-03

Name

RVA23

State

ratified

Ratification date

2023-04-03

3. RVA22 Profile Release

This profile release targets 64-bit application processors for markets requiring a high-degree of binary compatibility between compliant implementations.

RVA22 has 32 associated implementation-defined parameters across all its defined profiles.

3.1. RVA22 Description

This profile release is intended to be used for 64-bit application processors running rich OS stacks. Only user-mode and supervisor-mode profiles are specified in this release.

There is no machine-mode profile currently defined for this release. A machine-mode profile for application processors would only be used in specifying platforms for portable machine-mode software. Given the relatively low volume of portable M-mode software in this domain, the wide variety of potential M-mode code, and the very specific needs of each type of M-mode software, we are not specifying individual M-mode ISA requirements in this release.

Only XLEN=64 application processor profiles are currently defined. It would be possible to also define very similar XLEN=32 variants.

3.2. RVA22U64 Profile

The RVA22U64 profile specifies the ISA features available to user-mode execution environments in 64-bit applications processors. This is the most important profile within application processors in terms of the amount of software that targets this profile.

3.2.1. Mandatory Extensions

The RVA22U64 Profile has 27 mandatory extensions.

- **A** Atomic instructions

  Version ~> 2.1

- **B** Bit Manipulation

  Version ~> 1.0.0

- **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

- **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.

- **M** Integer multiply and divide

  Version ~> 2.0

- **U** User-mode privilege level

  Version ~> 2.0

- **Za128rs** Reservation set size of at most 128 bytes

  Version ~> 1.0

  > Za128rs is a profile-defined extension introduced with RVA20. The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

- **Za64rs** Reservation set size of at most 64 bytes

  Version ~> 1.0

  > Reservation sets are contiguous, naturally aligned, and a maximum of 64 bytes

- **Zba** Address generation

  Version ~> 1.0

- **Zbb** Basic bit-manipulation

  Version ~> 1.0

- **Zbs** Single-bit instructions

  Version ~> 1.0

- **Zfhmin** Minimal half-precision floating-point

  Version ~> 1.0

  > Zfhmin is a small extension that adds support to load/store and convert IEEE 754 half-precision numbers to and from the IEEE 754 single-precision format. The hardware cost for this extension is low, and mandating the extension avoids adding an option to the profile.

- **Zic64b** 64-byte cache blocks

  Version = 1.0

  > This is a new extension name for this feature. While the general RISC-V specifications are agnostic to cache block size, selecting a common cache block size simplifies the specification and use of the following cache-block extensions within the application processor profile. Software does not have to query a discovery mechanism and/or provide dynamic dispatch to the appropriate code. We choose 64 bytes at it is effectively an industry standard. Implementations may use longer cache blocks to reduce tag cost provided they use 64-byte sub-blocks to remain compatible. Implementations may use shorter cache blocks provided they sequence cache operations across the multiple cache blocks comprising a 64-byte block to remain compatible.

- **Zicbom** Cache-block management instructions

  Version ~> 1.0

- **Zicbop** Cache-block prefetch

  Version ~> 1.0

  > As with other HINTS, the inclusion of prefetches in the mandatory set of extensions indicates that software should generate these instructions where they are expected to be useful, and hardware is expected to exploit that information.

- **Zicboz** Cache-block zero instruction

  Version ~> 1.0

- **Ziccamoa** Main memory supports all atomics in A extension

  Version ~> 1.0

  > Ziccamo is a profile-defined extension introduced with RVA20.

- **Ziccif** Main memory supports instruction fetch with atomicity requirement

  Version ~> 1.0

  > Ziccif is a profile-defined extension introduced with RVA20. The fetch atomicity requirement facilitates runtime patching of aligned instructions.

- **Zicclsm** Main memory supports misaligned loads/stores

  Version ~> 1.0

  > Zicclsm is a profile-defined extension introduced with RVA20. This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

- **Ziccrse** Main memory supports forward progress on LR/SC sequences

  Version ~> 1.0

  > Ziccrse is a profile-defined extension introduced with RVA20.

- **Zicntr** Base Counters and Timers

  Version ~> 2.0

- **Zicsr** Control and status register instructions

  Version ~> 2.0

- **Zihintpause** Pause Hint

  Version ~> 2.0

  > While the pause instruction is a HINT can be implemented as a NOP and hence trivially supported by hardware implementers, its inclusion in the mandatory extension list signifies that software should use the instruction whenever it would make sense and that implementers are expected to exploit this information to optimize hardware execution.

- **Zihpm** Hardware Performance Counters

  Version ~> 2.0

  > The number of counters is platform-specific.

- **Zkt** Data-independent execution latency

  Version ~> 1.0

  > Zkt requires a certain subset of integer instructions execute with data-independent latency. Mandating this feature enables portable libraries for safe basic cryptographic operations. It is expected that application processors will naturally have this property and so implementation cost is low, if not zero, in most systems that would support RVA22.

3.2.2. Optional Extensions

The RVA22U64 Profile has 7 optional extensions.

- **V** Vector Operations

  Version ~> 1.0

  > The smaller vector extensions (Zve32f, Zve32x, Zve64d, Zve64f, Zve64x) are not provided as separately supported profile options. The full V extension is specified as the only supported profile option.
  >
  > A future profile might mandate V.

- **Zca** C instructions excluding floating-point loads/stores

  Version ~> 1.0

- **Zcd** Compressed double-precision floating-point loads/stores

  Version ~> 1.0

- **Zcf** Compressed single-precision floating-point loads/stores

  Version ~> 1.0

- **Zfh** Half-precision floating point

  Version ~> 1.0

  > A future profile might mandate V.

- **Zkn** NIST Algorithm Suite

  Version ~> 1.0

- **Zks** ShangMi Algorithm Suite

  Version ~> 1.0

> The scalar crypto extensions are expected to be superseded by vector crypto standards in future profiles, and the scalar extensions may be removed as supported options once vector crypto is present.

> The smaller component scalar crypto extensions (Zbc, Zbkb, Zbkc, Zbkx, Zknd, Zkne, Zknh, Zksed, Zksh) are not provided as separate options in the profile. Profile implementers should provide all of the instructions in a given algorithm suite as part of the Zkn or Zks supported options.

> Access to the entropy source (Zkr) in a system is usually carefully controlled. While the design supports unprivileged access to the entropy source, this is unlikely to be commonly used in an application processor, and so Zkr was not added as a profile option. This also means the roll-up Zk was not added as a profile option.

> The Zfinx, Zdinx, Zhinx, Zhinxmin extensions are incompatible with the profile mandates to support the F and D extensions.

3.2.3. Implementation-dependencies

RVA22U64 has 9 associated implementation-defined parameters.

MCOUNTENABLE_EN
: Indicates which counters can be delegated via mcounteren.

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

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

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

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

UXLEN
: Set of XLENs supported in U-mode. When both 32 and 64 are supported, SXLEN can be changed, via mstatus.UXL, between 32 and 64.

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

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

FORCE_UPGRADE_CBO_INVAL_TO_FLUSH

When true, an implementation prohibits setting menvcfg.CBIE == 11 such that all cbo.inval instructions either trap (when menvcfg.CBIE == '00') or flush (when menvcfg.CBIE == '01').

When false, an implementation allows a true INVAL operation for cbo.inval, and thus supports the setting menvcfg.CBIE == 11.

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

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. RVA22S64 Profile

The RVA22S64 profile specifies the ISA features available to a supervisor-mode execution environment in 64-bit applications processors. RVA22S64 is based on privileged architecture version 1.12.

3.3.1. Mandatory Extensions

The RVA22S64 Profile has 38 mandatory extensions.

- **A** Atomic instructions

  Version ~> 2.1

- **B** Bit Manipulation

  Version ~> 1.0.0

- **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

- **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.

- **M** Integer multiply and divide

  Version ~> 2.0

- **S** Supervisor mode

  Version ~> 1.12

- **Ssccptr** Cacheable and coherent main memory page table reads

  Version ~> 1.0

Ssccptr is a new extension name introduced with RVA20.

- **Sscounterenw** Support writeable enables for any supported counter

  Version ~> 1.0

Sstvala is a new extension name introduced with RVA22.

- **Sstvala** Supervisor Trap Value provides all needed values

  Version ~> 1.0

Sstvala is a new extension name introduced with RVA20.

- **Sstvecd** Direct exception vectoring

  Version ~> 1.0

Sstvecd is a new extension name introduced with RVA20.

- **Sv39** 39-bit virtual address translation (3 level)

  Version ~> 1.12

- **Svade** Exception on PTE A/D Bits

  Version ~> 1.0

Svbare is a new extension name introduced with RVA20.

It is subsequently defined in more detail with the ratification of Svadu.

- **Svbare** Bare virtual addressing

  Version ~> 1.0

Svbare is a new extension name introduced with RVA20.

- **Svinval** Fine-grained address-translation cache invalidation

  Version ~> 1.0

- **Svpbmt** Page-based memory types

  Version ~> 1.0

- **U** User-mode privilege level

  Version ~> 2.0

- **Za128rs** Reservation set size of at most 128 bytes

Version ~> 1.0

Za128rs is a profile-defined extension introduced with RVA20. The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

- **Za64rs** Reservation set size of at most 64 bytes

  Version ~> 1.0

  Reservation sets are contiguous, naturally aligned, and a maximum of 64 bytes

- **Zba** Address generation

  Version ~> 1.0

- **Zbb** Basic bit-manipulation

  Version ~> 1.0

- **Zbs** Single-bit instructions

  Version ~> 1.0

- **Zfhmin** Minimal half-precision floating-point

  Version ~> 1.0

  Zfhmin is a small extension that adds support to load/store and convert IEEE 754 half-precision numbers to and from the IEEE 754 single-precision format. The hardware cost for this extension is low, and mandating the extension avoids adding an option to the profile.

- **Zic64b** 64-byte cache blocks

  Version = 1.0

  This is a new extension name for this feature. While the general RISC-V specifications are agnostic to cache block size, selecting a common cache block size simplifies the specification and use of the following cache-block extensions within the application processor profile. Software does not have to query a discovery mechanism and/or provide dynamic dispatch to the appropriate code. We choose 64 bytes at it is effectively an industry standard. Implementations may use longer cache blocks to reduce tag cost provided they use 64-byte sub-blocks to remain compatible. Implementations may use shorter cache blocks provided they sequence cache operations across the multiple cache blocks comprising a 64-byte block to remain compatible.

- **Zicbom** Cache-block management instructions

  Version ~> 1.0

- **Zicbop** Cache-block prefetch

  Version ~> 1.0

  As with other HINTS, the inclusion of prefetches in the mandatory set of extensions indicates that software should generate these instructions where they are expected to be useful, and hardware is expected to exploit that information.

- **Zicboz** Cache-block zero instruction

  Version ~> 1.0

- **Ziccamoa** Main memory supports all atomics in A extension

  Version ~> 1.0

  Ziccamo is a profile-defined extension introduced with RVA20.

- **Ziccif** Main memory supports instruction fetch with atomicity requirement

  Version ~> 1.0

  Ziccif is a profile-defined extension introduced with RVA20. The fetch atomicity requirement facilitates runtime patching of aligned instructions.

- **Zicclsm** Main memory supports misaligned loads/stores

  Version ~> 1.0

Zicclsm is a profile-defined extension introduced with RVA20. This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

- **Ziccrse** Main memory supports forward progress on LR/SC sequences

  Version ~> 1.0

Ziccrse is a profile-defined extension introduced with RVA20.

- **Zicntr** Base Counters and Timers

  Version ~> 2.0

- **Zicsr** Control and status register instructions

  Version ~> 2.0

- **Zifencei** Instruction fence

  Version ~> 2.0

Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVA20 application processors. A new instruction-cache coherence mechanism is under development which might be added as an option in the future.

- **Zihintpause** Pause Hint

  Version ~> 2.0

While the pause instruction is a HINT can be implemented as a NOP and hence trivially supported by hardware implementers, its inclusion in the mandatory extension list signifies that software should use the instruction whenever it would make sense and that implementers are expected to exploit this information to optimize hardware execution.

- **Zihpm** Hardware Performance Counters

  Version ~> 2.0

The number of counters is platform-specific.

- **Zkt** Data-independent execution latency

  Version ~> 1.0

Zkt requires a certain subset of integer instructions execute with data-independent latency. Mandating this feature enables portable libraries for safe basic cryptographic operations. It is expected that application processors will naturally have this property and so implementation cost is low, if not zero, in most systems that would support RVA22.

3.3.2. Optional Extensions

The RVA22S64 Profile has 15 optional extensions.

- **Sha** Augmented hypervisor

  Version ~> 1.0

The set of augmented hypervisor extensions: H, Ssstateen, Shcounterenw, Shvstvala, Shtvala, Shvstvecd, Shvsatpa, Shgatpa. The Sha extension name was introduced in RVA23/RVB23 and wasn't in the ratified RVA22 profile but has since been added retroactively as a documentation-only change (doesn't affect requirements or functionality).

- **Sscofpmf** Counter Overflow and Privilege Mode Filtering

  Version ~> 1.0

Platforms may choose to mandate the presence of Sscofpmf.

- **Sstc** Supervisor-mode timer interrupts

  Version ~> 1.0

Sstc was not made mandatory in RVA22S64 as it is a more disruptive change affecting system-level architecture, and will take longer for implementations to adopt. It is expected to be made mandatory in the next profile release.

- **Ssu64xl** 64-bit UXLEN

  Version ~> 1.0

  > Ssu64xl is a new extension name introduced with RVA20.

- **Sv48** 48-bit virtual address translation (4 level)

  Version ~> 1.12

- **Sv57** 57-bit virtual address translation (5 level)

  Version ~> 1.12

- **Svnapot** Naturally Aligned Power-of-Two Translation Contiguity

  Version ~> 1.0

  > It is expected that Svnapot will be mandatory in the next profile release.

- **V** Vector Operations

  Version ~> 1.0

  > The smaller vector extensions (Zve32f, Zve32x, Zve64d, Zve64f, Zve64x) are not provided as separately supported profile options. The full V extension is specified as the only supported profile option.
  >
  > A future profile might mandate V.

- **Zca** C instructions excluding floating-point loads/stores

  Version ~> 1.0

- **Zcd** Compressed double-precision floating-point loads/stores

  Version ~> 1.0

- **Zcf** Compressed single-precision floating-point loads/stores

  Version ~> 1.0

- **Zfh** Half-precision floating point

  Version ~> 1.0

  > A future profile might mandate V.

- **Zkn** NIST Algorithm Suite

  Version ~> 1.0

- **Zkr** Entropy Source

  Version ~> 1.0

  > Technically, Zk is also a privileged-mode option capturing that Zkr, Zkn, and Zkt are all implemented. However, the Zk rollup is less descriptive than specifying the individual extensions explicitly.

- **Zks** ShangMi Algorithm Suite

  Version ~> 1.0

3.3.3. Implementation-dependencies

RVA22S64 has 32 associated implementation-defined parameters.

ASID_WIDTH

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

MSTATUS_TVM_IMPLEMENTED

Whether or not mstatus.TVM is implemented.

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

REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION

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

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

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT

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

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

REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT

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

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

REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT

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

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

REPORT_VA_IN_STVAL_ON_BREAKPOINT

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

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

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT

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

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED

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

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

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT

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

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

REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT

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

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

REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED

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

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

REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT

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

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

REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT

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

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

REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED

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

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

REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT

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

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

SATP_MODE_BARE

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

STVAL_WIDTH

The number of implemented bits in stval.

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

STVEC_MODE_DIRECT

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

STVEC_MODE_VECTORED

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

SXLEN

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

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

S_MODE_ENDIANNESS

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

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

TRAP_ON_ECALL_FROM_S

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

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

MCOUNTENABLE_EN

Indicates which counters can be delegated via mcounteren.

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

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

TRAP_ON_ECALL_FROM_U

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

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

UXLEN

Set of XLENs supported in U-mode. When both 32 and 64 are supported, SXLEN can be changed, via mstatus.UXL, between 32 and 64.

U_MODE_ENDIANNESS

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

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

FORCE_UPGRADE_CBO_INVAL_TO_FLUSH

When true, an implementation prohibits setting menvcfg.CBIE == 11 such that all cbo.inval instructions either trap (when menvcfg.CBIE == '00') or flush (when menvcfg.CBIE == '01').

When false, an implementation allows a true INVAL operation for cbo.inval, and thus supports the setting menvcfg.CBIE == 11.

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

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. Apps Processor Profile Releases

The Apps Processor processor kind has 4 processor profile releases that reference a total of 89 extensions.

Table 1. Extension Presence

Extension	RVA20	RVA22	RVA23	RVB23
A	mandatory	mandatory	mandatory	mandatory
B	-	mandatory	mandatory	mandatory
C	mandatory	mandatory	mandatory	mandatory
D	mandatory	mandatory	mandatory	mandatory
F	mandatory	mandatory	mandatory	mandatory
I	mandatory	mandatory	mandatory	mandatory
M	mandatory	mandatory	mandatory	mandatory
S	mandatory	mandatory	mandatory	mandatory
Sdtrig	-	-	optional	optional
Sha	-	optional	mandatory	optional
Sm	-	-	-	-
Smpmp	-	-	-	-
Ssccptr	mandatory	mandatory	mandatory	mandatory
Sscofpmf	-	optional	mandatory	mandatory
Sscounterenw	-	mandatory	mandatory	mandatory
Ssnpm	-	-	mandatory	optional
Sspm	-	-	optional	optional
Ssstrict	-	-	optional	optional
Sstc	-	optional	mandatory	mandatory
Sstvala	mandatory	mandatory	mandatory	mandatory
Sstvecd	mandatory	mandatory	mandatory	mandatory
Ssu64xl	optional	optional	mandatory	mandatory
Supm	-	-	mandatory	optional
Sv39	mandatory	mandatory	mandatory	mandatory
Sv48	optional	optional	optional	optional
Sv57	-	optional	optional	optional
Svade	mandatory	mandatory	mandatory	mandatory
Svadu	-	-	optional	optional
Svbare	mandatory	mandatory	mandatory	mandatory
Svinval	-	mandatory	mandatory	mandatory
Svnapot	-	optional	mandatory	mandatory
Svpbmt	-	mandatory	mandatory	mandatory
Svvptc	-	-	optional	optional
U	mandatory	mandatory	mandatory	mandatory
V	-	optional	mandatory	optional
Za128rs	mandatory	mandatory	mandatory	mandatory
Za64rs	-	mandatory	mandatory	mandatory
Zabha	-	-	optional	optional
Zacas	-	-	optional	optional
Zama16b	-	-	optional	optional
Zawrs	-	-	mandatory	mandatory
Zba	-	mandatory	mandatory	mandatory
Zbb	-	mandatory	mandatory	mandatory

Extension	RVA20	RVA22	RVA23	RVB23
Zbc	-	-	optional	optional
Zbs	-	mandatory	mandatory	mandatory
Zca	optional	optional	optional	optional
Zcb	-	-	mandatory	mandatory
Zcd	optional	optional	optional	optional
Zcf	optional	optional	optional	optional
Zcmop	-	-	mandatory	mandatory
Zfa	-	-	mandatory	mandatory
Zfbfmin	-	-	optional	optional
Zfh	-	optional	optional	optional
Zfhmin	-	mandatory	mandatory	optional
Zic64b	-	mandatory	mandatory	mandatory
Zicbom	-	mandatory	mandatory	mandatory
Zicbop	-	mandatory	mandatory	mandatory
Zicboz	-	mandatory	mandatory	mandatory
Ziccamoa	mandatory	mandatory	mandatory	mandatory
Ziccamoc	-	-	optional	optional
Ziccif	mandatory	mandatory	mandatory	mandatory
Zicclsm	mandatory	mandatory	mandatory	mandatory
Ziccrse	mandatory	mandatory	mandatory	mandatory
Zicfilp	-	-	optional	optional
Zicfiss	-	-	optional	optional
Zicntr	mandatory	mandatory	mandatory	mandatory
Zicond	-	-	mandatory	mandatory
Zicsr	mandatory	mandatory	mandatory	mandatory
Zifencei	mandatory	mandatory	mandatory	mandatory
Zihintntl	-	-	mandatory	mandatory
Zihintpause	-	mandatory	mandatory	mandatory
Zihpm	optional	mandatory	mandatory	mandatory
Zimop	-	-	mandatory	mandatory
Zkn	-	optional	-	optional
Zkr	-	optional	optional	optional
Zks	-	optional	-	optional
Zkt	-	mandatory	mandatory	mandatory
Zvbb	-	-	mandatory	optional
Zvbc	-	-	optional	optional
Zvfbfmin	-	-	optional	optional
Zvfbfwma	-	-	optional	optional
Zvfh	-	-	optional	optional
Zvfhmin	-	-	mandatory	optional
Zvkg	-	-	-	optional
Zvknc	-	-	-	optional
Zvkng	-	-	optional	optional
Zvksc	-	-	-	optional
Zvksg	-	-	optional	optional
Zvkt	-	-	mandatory	optional

A.2. RVA Profile Releases

The RVA Profile Family has 3 releases that reference a total of 86 extensions.

Table 2. Extension Presence

Extension	**RVA20**	**RVA22**	**RVA23**
A	mandatory	mandatory	mandatory
B	-	mandatory	mandatory
C	mandatory	mandatory	mandatory
D	mandatory	mandatory	mandatory
F	mandatory	mandatory	mandatory
I	mandatory	mandatory	mandatory
M	mandatory	mandatory	mandatory
S	mandatory	mandatory	mandatory
Sdtrig	-	-	optional
Sha	-	optional	mandatory
Sm	-	-	-
Smpmp	-	-	-
Ssccptr	mandatory	mandatory	mandatory
Sscofpmf	-	optional	mandatory
Sscounterenw	-	mandatory	mandatory
Ssnpm	-	-	mandatory
Sspm	-	-	optional
Ssstrict	-	-	optional
Sstc	-	optional	mandatory
Sstvala	mandatory	mandatory	mandatory
Sstvecd	mandatory	mandatory	mandatory
Ssu64xl	optional	optional	mandatory
Supm	-	-	mandatory
Sv39	mandatory	mandatory	mandatory
Sv48	optional	optional	optional
Sv57	-	optional	optional
Svade	mandatory	mandatory	mandatory
Svadu	-	-	optional
Svbare	mandatory	mandatory	mandatory
Svinval	-	mandatory	mandatory
Svnapot	-	optional	mandatory
Svpbmt	-	mandatory	mandatory
Svvptc	-	-	optional
U	mandatory	mandatory	mandatory
V	-	optional	mandatory
Za128rs	mandatory	mandatory	mandatory
Za64rs	-	mandatory	mandatory
Zabha	-	-	optional
Zacas	-	-	optional
Zama16b	-	-	optional
Zawrs	-	-	mandatory
Zba	-	mandatory	mandatory
Zbb	-	mandatory	mandatory
Zbc	-	-	optional
Zbs	-	mandatory	mandatory
Zca	optional	optional	optional
Zcb	-	-	mandatory
Zcd	optional	optional	optional

Extension	RVA20	RVA22	RVA23
Zcf	optional	optional	optional
Zcmop	-	-	mandatory
Zfa	-	-	mandatory
Zfbfmin	-	-	optional
Zfh	-	optional	optional
Zfhmin	-	mandatory	mandatory
Zic64b	-	mandatory	mandatory
Zicbom	-	mandatory	mandatory
Zicbop	-	mandatory	mandatory
Zicboz	-	mandatory	mandatory
Ziccamoa	mandatory	mandatory	mandatory
Ziccamoc	-	-	optional
Ziccif	mandatory	mandatory	mandatory
Zicclsm	mandatory	mandatory	mandatory
Ziccrse	mandatory	mandatory	mandatory
Zicfilp	-	-	optional
Zicfiss	-	-	optional
Zicntr	mandatory	mandatory	mandatory
Zicond	-	-	mandatory
Zicsr	mandatory	mandatory	mandatory
Zifencei	mandatory	mandatory	mandatory
Zihintntl	-	-	mandatory
Zihintpause	-	mandatory	mandatory
Zihpm	optional	mandatory	mandatory
Zimop	-	-	mandatory
Zkn	-	optional	-
Zkr	-	optional	optional
Zks	-	optional	-
Zkt	-	mandatory	mandatory
Zvbb	-	-	mandatory
Zvbc	-	-	optional
Zvfbfmin	-	-	optional
Zvfbfwma	-	-	optional
Zvfh	-	-	optional
Zvfhmin	-	-	mandatory
Zvkng	-	-	optional
Zvksg	-	-	optional
Zvkt	-	-	mandatory

A.3. RVA22 Profiles

The RVA22 Profile Release has 2 profiles that reference a total of 53 extensions.

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

Table 3. Extension Presence

Extension	RVA22U64	RVA22S64
A	mandatory	mandatory
B	mandatory	mandatory
C	mandatory	mandatory
D	mandatory	mandatory

Extension	RVA22U64	RVA22S64
F	mandatory	mandatory
I	mandatory	mandatory
M	mandatory	mandatory
S	-	mandatory
Sha	-	option
Ssccptr	-	mandatory
Sscofpmf	-	option
Sscounterenw	-	mandatory
Sstc	-	option
Sstvala	-	mandatory
Sstvecd	-	mandatory
Ssu64xl	-	option
Sv39	-	mandatory
Sv48	-	option
Sv57	-	option
Svade	-	mandatory
Svbare	-	mandatory
Svinval	-	mandatory
Svnapot	-	option
Svpbmt	-	mandatory
U	mandatory	mandatory
V	option	option
Za128rs	mandatory	mandatory
Za64rs	mandatory	mandatory
Zba	mandatory	mandatory
Zbb	mandatory	mandatory
Zbs	mandatory	mandatory
Zca	option	option
Zcd	option	option
Zcf	option	option
Zfh	option	option
Zfhmin	mandatory	mandatory
Zic64b	mandatory	mandatory
Zicbom	mandatory	mandatory
Zicbop	mandatory	mandatory
Zicboz	mandatory	mandatory
Ziccamoa	mandatory	mandatory
Ziccif	mandatory	mandatory
Zicclsm	mandatory	mandatory
Ziccrse	mandatory	mandatory
Zicntr	mandatory	mandatory
Zicsr	mandatory	mandatory
Zifencei	-	mandatory
Zihintpause	mandatory	mandatory
Zihpm	mandatory	mandatory
Zkn	option	option
Zkr	-	option
Zks	option	option
Zkt	mandatory	mandatory

Appendix B: Extension Details

B.1. Extension A

Long Name: Atomic instructions
Version Requirement: ~> 2.1
A Extension Presence

profile	v2.1.0
RVA22U64	mandatory
RVA22S64	mandatory

B.1.1. Available Versions

	Version 2.1.0
State	ratified
Ratification date	2019-12

B.1.2. 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.1.3. 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.2. Extension B

Long Name: Bit Manipulation
Version Requirement: ~> 1.0.0
B Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.2.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2024-04
Ratification document	drive.google.com/file/d/1SgLoasaBjs5WboQMaU3wpHkjUwV71UZn/view

B.2.2. Synopsis

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

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

B.3. Extension C

Long Name: Compressed instructions
Version Requirement: ~> 2.0
C Extension Presence

profile	**v2.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.3.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-12

B.3.2. Synopsis

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

B.3.3. Overview

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

B.3.4. Compressed Instruction Formats

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

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

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

The standard RISC-V calling convention maps the most frequently used floating-point registers to registers `f8` to `f15`, which allows the

> *same register decompression decoding as for integer register numbers.*

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

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

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

Table 4. Compressed 16-bit RVC instruction formats

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

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

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

B.4. Extension D

Long Name: Double-precision floating-point
Version Requirement: ~> 2.2
D Extension Presence

profile	**v2.2.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.4.1. Available Versions

	Version 2.2.0
State	ratified
Ratification date	2019-12
Changes	• Define NaN-boxing scheme, changed definition of FMAX and FMIN

B.4.2. 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.4.3. 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.4.4.

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.4.4. 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.4.5. Instructions

The following 26 instructions are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

fadd.d	**Floating-Point Add Double-Precision**
fclass.d	**Floating-Point Classify Double-Precision**
fcvt.d.s	**Floating-Point Convert Single-Precision to Double-Precision**
fcvt.d.w	**Floating-Point Convert Word to Double-Precision**
fcvt.d.wu	**Floating-Point Convert Unsigned Word to Double-Precision**
fcvt.s.d	**Floating-Point Convert Double-Precision to Single-Precision**
fcvt.w.d	**Floating-Point Convert Double-Precision to Word**
fcvt.wu.d	**Floating-Point Convert Double-Precision to Unsigned Word**
fdiv.d	**Floating-Point Divide Double-Precision**
feq.d	**Floating-Point Equal Double-Precision**
fld	**Floating-Point Load Double-Precision**
fle.d	**Floating-Point Less Than or Equal Double-Precision**
flt.d	**Floating-Point Less Than Double-Precision**
fmadd.d	**Floating-Point Multiply-Add Double-Precision**
fmax.d	**Floating-Point Maximum-Number Double-Precision**
fmin.d	**Floating-Point Minimum-Number Double-Precision**

fmsub.d	**Floating-Point Multiply-Subtract Double-Precision**
fmul.d	**Floating-Point Multiply Double-Precision**
fnmadd.d	**Floating-Point Negate-Multiply-Add Double-Precision**
fnmsub.d	**Floating-Point Negate-Multiply-Subtract Double-Precision**
fsd	**Floating-Point Store Double-Precision**
fsgnj.d	**Floating-Point Sign-Inject Double-Precision**
fsgnjn.d	**Floating-Point Sign-Inject Negate Double-Precision**
fsgnjx.d	**Floating-Point Sign-Inject XOR Double-Precision**
fsqrt.d	**Floating-Point Square Root Double-Precision**
fsub.d	**Floating-Point Subtract Double-Precision**

B.5. Extension F

Long Name: Single-precision floating-point
Version Requirement: ~> 2.2
F Extension Presence

profile	**v2.2.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.5.1. Available Versions

	Version 2.2.0
State	ratified
Ratification date	2019-12
Changes	• Define NaN-boxing scheme, changed definition of FMAX and FMIN

B.5.2. 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.5.3. 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	

FLEN-1		0
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 RVA22ProfileRelease.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`)

Rounding Mode	Mnemonic	Meaning
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.5.4. 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.5.5. 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.5.6. Instructions

The following 26 instructions are added by extension version 2.2.0 (the minimum version of this extension that satifies the extension requirement).

fadd.s	**Floating-Point Add Single-Precision**
fclass.s	**Floating-Point Classify Single-Precision**
fcvt.s.w	**Floating-Point Convert Word to Single-Precision**
fcvt.s.wu	**Floating-Point Convert Unsigned Word to Single-Precision**

fcvt.w.s	**Floating-Point Convert Single-Precision to Word**
fcvt.wu.s	**Floating-Point Convert Single-Precision to Unsigned Word**
fdiv.s	**Floating-Point Divide Single-Precision**
feq.s	**Floating-Point Equal Single-Precision**
fle.s	**Floating-Point Less Than or Equal Single-Precision**
flt.s	**Floating-Point Less Than Single-Precision**
flw	**Floating-Point Load Single-Precision**
fmadd.s	**Floating-Point Multiply-Add Single-Precision**
fmax.s	**Floating-Point Maximum-Number Single-Precision**
fmin.s	**Floating-Point Minimum-Number Single-Precision**
fmsub.s	**Floating-Point Multiply-Subtract Single-Precision**
fmul.s	**Floating-Point Multiply Single-Precision**
fmv.w.x	**Floating-Point Move Single-Precision Word from Integer Register**
fmv.x.w	**Floating-Point Move Single-Precision Word to Integer Register**
fnmadd.s	**Floating-Point Negate-Multiply-Add Single-Precision**
fnmsub.s	**Floating-Point Negate-Multiply-Subtract Single-Precision**
fsgnj.s	**Floating-Point Sign-Inject Single-Precision**
fsgnjn.s	**Floating-Point Sign-Inject Negate Single-Precision**
fsgnjx.s	**Floating-Point Sign-Inject XOR Single-Precision**
fsqrt.s	**Floating-Point Square Root Single-Precision**
fsub.s	**Floating-Point Subtract Single-Precision**
fsw	**Floating-Point Store Single-Precision**

B.5.7. 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.6. Extension I

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

profile	**v2.1.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.6.1. Available Versions

	Version 2.1.0
State	ratified
Ratification date	2019-06
Changes	• ratified RVWMO memory model and exclusion of FENCE.I, counters, and CSR instructions that were in previous base ISA

B.6.2. 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.6.3. Instructions

The following 43 instructions are added by extension version 2.1.0 (the minimum version of this extension that satifies the extension requirement).

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

sub	**Subtract**
sw	**Store word**
xor	**Exclusive Or**
xori	**Exclusive Or immediate**

B.7. Extension M

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

profile	**v2.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.7.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-12

B.7.2. Synopsis

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

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

B.7.3. Instructions

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

div	**Signed division**
divu	**Unsigned division**
mul	**Signed multiply**
mulh	**Signed multiply high**
mulhsu	**Signed/unsigned multiply high**
mulhu	**Unsigned multiply high**
rem	**Signed remainder**
remu	**Unsigned remainder**

B.8. Extension S

Long Name: Supervisor mode
Version Requirement: ~> 1.12
S Extension Presence

profile	**v1.11.0**	**v1.12.0**	**v1.13.0**
RVA22U64	-	-	-
RVA22S64	mandatory	mandatory	mandatory

B.8.1. Available Versions

	Version 1.11.0
State	ratified
Ratification date	2019-06

	Version 1.12.0
State	ratified

Version 1.12.0	
Ratification date	2021-12

Version 1.13.0	
State	ratified
Ratification date	2024-10

B.8.2. Synopsis

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

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

B.8.3. Instructions

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

sfence.vma	**Supervisor memory-management fence**
sret	**Supervisor Mode Return from Trap**

B.8.4. CSRs

The following 10 CSRs are added by extension version 1.11.0 (the minimum version of this extension that satifies the extension requirement).

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

B.8.5. Parameters

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

ASID_WIDTH

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

MSTATUS_TVM_IMPLEMENTED

Whether or not mstatus.TVM is implemented.

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

REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION

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

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

REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT

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

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

REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT

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

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

REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT

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

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

REPORT_VA_IN_STVAL_ON_BREAKPOINT

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

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

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT

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

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED

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

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

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

REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT

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

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

REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT

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

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

REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED

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

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

REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT

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

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

REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT

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

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

REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED

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

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

REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT

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

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

SATP_MODE_BARE

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

STVAL_WIDTH

The number of implemented bits in stval.

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

STVEC_MODE_DIRECT

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

STVEC_MODE_VECTORED

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

SXLEN

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

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

S_MODE_ENDIANNESS

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

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

TRAP_ON_ECALL_FROM_S

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

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

B.9. Extension Sha

Long Name: Augmented hypervisor
Version Requirement: ~> 1.0
Sha Extension Presence

profile	**v1.0.0**
RVA22U64	-
RVA22S64	optional

B.9.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2024-10

B.9.2. Synopsis

Sha comprises the following extensions:

- **H** The hypervisor extension.
- **Ssstateen** Supervisor-mode view of the state-enable extension. The supervisor-mode (`sstateen0-3`) and hypervisor-mode (`hstateen0-3`) state-enable registers must be provided.
- **Shcounterenw** For any `hpmcounter` that is not read-only zero, the corresponding bit in hcounteren must be writable.
- **Shvstvala** vstval must be written in all cases described above for stval.
- **Shtvala** htval must be written with the faulting guest physical address in all circumstances permitted by the ISA.
- **Shvstvecd** vstvec.MODE must be capable of holding the value 0 (Direct). When vstvec.MODE=Direct, vstvec.BASE must be capable of holding any valid four-byte-aligned address.
- **Shvsatpa** All translation modes supported in satp must be supported in vsatp.
- **Shgatpa** For each supported virtual memory scheme SvNN supported in satp, the corresponding hgatp SvNNx4 mode must be supported. The hgatp mode Bare must also be supported.

Sha is a new profile-defined extension that captures the full set of features that are mandated to be supported along with the H extension. There is no change to the features added by including the hypervisor extension in a profile—the new name is solely to simplify the text of the profiles. The definition has been added to the RVA22 profile text, where the hypervisor extension was first added, and will be added to the hypervisor section of the combined ISA manual.

The set of augmented hypervisor extensions: H, Ssstateen, Shcounterenw, Shvstvala, Shtvala, Shvstvecd, Shvsatpa, Shgatpa. The Sha extension name was introduced in RVA23/RVB23 and wasn't in the ratified RVA22 profile but has since been added retroactively as a documentation-only change (doesn't affect requirements or functionality).

B.10. Extension Ssccptr

Long Name: Cacheable and coherent main memory page table reads
Version Requirement: ~> 1.0
Ssccptr Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

B.10.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03
Ratification document	github.com/riscv/riscv-profiles/releases/tag/v1.0

B.10.2. Synopsis

Main memory regions with both the cacheability and coherence PMAs must support hardware page-table reads.

This extension was ratified with the RVA20 profiles.

Ssccptr is a new extension name introduced with RVA20.

B.11. Extension Sscofpmf

Long Name: Counter Overflow and Privilege Mode Filtering
Version Requirement: ~> 1.0
Sscofpmf Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

B.11.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-08
Ratification document	drive.google.com/file/d/1KcjgbLM5L1ZKY8934aJl8aQwGlMz6Cbo/view?usp=drive_link

B.11.2. Synopsis

Counter Overflow and Privilege Mode Filtering

Platforms may choose to mandate the presence of Sscofpmf.

B.11.3. CSRs

The following 1 CSRs are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
scountovf	Supervisor Count Overflow	0xda0	S

B.12. Extension Sscounterenw

Long Name: Support writeable enables for any supported counter
Version Requirement: ~> 1.0
Sscounterenw Extension Presence

profile	v1.0.0
RVA22U64	-

profile	v1.0.0
RVA22S64	mandatory

B.12.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2023-08
Ratification document	drive.google.com/file/d/1KcjgbLM5L1ZKY8934aJl8aQwGlMz6Cbo/view?usp=drive_link

B.12.2. Synopsis

For any hpmcounter that is not read-only zero, the corresponding bit in scounteren must be writable.

This extension was ratified with the RVA22 profiles.

Sstvala is a new extension name introduced with RVA22.

B.13. Extension Sstc

Long Name: Supervisor-mode timer interrupts
Version Requirement: ~> 1.0
Sstc Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

B.13.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2024-03
Ratification document	drive.google.com/file/d/1m84Re2yK8m_vbW7TspvevCDR82MOBaSX/view?usp=drive_link

B.13.2. Synopsis

Supervisor-mode timer interrupts

Sstc was not made mandatory in RVA22S64 as it is a more disruptive change affecting system-level architecture, and will take longer for implementations to adopt. It is expected to be made mandatory in the next profile release.

B.14. Extension Sstvala

Long Name: Supervisor Trap Value provides all needed values
Version Requirement: ~> 1.0
Sstvala Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

B.14.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2023-03
Ratification document	github.com/riscv/riscv-profiles/releases/tag/v1.0

B.14.2. Synopsis

stval must be written with the faulting virtual address for load, store, and instruction page-fault, access-fault, and misaligned exceptions, and for

breakpoint exceptions other than those caused by execution of the ebreak or `c.ebreak instructions.

For virtual-instruction and illegal-instruction exceptions, stval must be written with the faulting instruction.

This extension was ratified with the RVA20 profiles.

Sstvala is a new extension name introduced with RVA20.

B.15. Extension Sstvecd

Long Name: Direct exception vectoring
Version Requirement: ~> 1.0
Sstvecd Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

B.15.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03
Ratification document	github.com/riscv/riscv-profiles/releases/tag/v1.0

B.15.2. Synopsis

stvec.MODE must be capable of holding the value 0 (Direct). When stvec.MODE=Direct, stvec.BASE must be capable of holding any valid four-byte-aligned address.

Sstvecd is a new extension name introduced with RVA20.

B.16. Extension Ssu64xl

Long Name: 64-bit UXLEN
Version Requirement: ~> 1.0
Ssu64xl Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	optional

B.16.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.16.2. Synopsis

sstatus.UXL must be capable of holding the value 2 (i.e., UXLEN=64 must be supported).

This extension is defined by RVA22.

Ssu64xl is a new extension name introduced with RVA20.

B.17. Extension Sv39

Long Name: 39-bit virtual address translation (3 level)
Version Requirement: ~> 1.12
Sv39 Extension Presence

profile	v1.11.0	v1.12.0	v1.13.0
RVA22U64	-	-	-
RVA22S64	mandatory	mandatory	mandatory

B.17.1. Available Versions

Version 1.11.0	
State	ratified
Ratification date	2019-05

Version 1.12.0	
State	ratified
Ratification date	2021-12
Ratification document	github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

Version 1.13.0	
State	ratified
Ratification date	2024-10

B.17.2. Synopsis

39-bit virtual address translation (3 level)

B.18. Extension Sv48

Long Name: 48-bit virtual address translation (4 level)
Version Requirement: ~> 1.12
Sv48 Extension Presence

profile	v1.11.0	v1.12.0	v1.13.0
RVA22U64	-	-	-
RVA22S64	optional	optional	optional

B.18.1. Available Versions

Version 1.11.0	
State	ratified
Ratification date	2019-05

Version 1.12.0	
State	ratified
Ratification date	2021-12
Ratification document	github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

Version 1.13.0	
State	ratified
Ratification date	2024-10

B.18.2. Synopsis

48-bit virtual address translation (4 level)

B.19. Extension Sv57

Long Name: 57-bit virtual address translation (5 level)
Version Requirement: ~> 1.12
Sv57 Extension Presence

profile	v1.11.0	v1.12.0	v1.13.0
RVA22U64	-	-	-

profile	v1.11.0	v1.12.0	v1.13.0
RVA22S64	optional	optional	optional

B.19.1. Available Versions

Version 1.11.0	
State	ratified

Version 1.12.0	
State	ratified
Ratification date	2021-12
Ratification document	github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

Version 1.13.0	
State	ratified
Ratification date	2024-10

B.19.2. Synopsis

57-bit virtual address translation (5 level)

B.20. Extension Svade

Long Name: Exception on PTE A/D Bits
Version Requirement: ~> 1.0
Svade Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

B.20.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2023-11
Ratification document	github.com/riscvarchive/riscv-svadu/releases/download/v1.0/riscv-svadu.pdf

B.20.2. Synopsis

The Svade extension indicates that hardware does **not** update the A/D bits of a page table during a page walk. Rather, encountering a PTE with the A bit clear or the D bit clear when an operation is a write will cause a Page Fault.

Svbare is a new extension name introduced with RVA20.

It is subsequently defined in more detail with the ratification of Svadu.

B.21. Extension Svbare

Long Name: Bare virtual addressing
Version Requirement: ~> 1.0
Svbare Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

B.21.1. Available Versions

Version 1.0.0	
State	ratified

	Version 1.0.0
Ratification date	2023-03

B.21.2. Synopsis

This extension mandates that the satp mode Bare must be supported.

This extension was ratified as part of the RVA22 profile.

Svbare is a new extension name introduced with RVA20.

B.22. Extension Svinval

Long Name: Fine-grained address-translation cache invalidation
Version Requirement: ~> 1.0
Svinval Extension Presence

profile	v1.0.0
RVA22U64	-
RVA22S64	mandatory

B.22.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-11

B.22.2. Synopsis

The Svinval extension splits sfence.vma, `hfence.vma`, and hfence.gvma instructions into finer-grained invalidation and ordering operations that can be more efficiently batched or pipelined on certain classes of high-performance implementation.

The sinval.vma instruction invalidates any address-translation cache entries that an sfence.vma instruction with the same values of rs1 and rs2 would invalidate. However, unlike sfence.vma, sinval.vma instructions are only ordered with respect to sfence.vma, sfence.w.inval, and sfence.inval.ir instructions as defined below.

The sfence.w.inval instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before subsequent sinval.vma instructions executed by the same hart. The sfence.inval.ir instruction guarantees that any previous sinval.vma instructions executed by the current hart are ordered before subsequent implicit references by that hart to the memory-management data structures.

When executed in order (but not necessarily consecutively) by a single hart, the sequence sfence.w.inval, sinval.vma, and sfence.inval.ir has the same effect as a hypothetical sfence.vma instruction in which:

- the values of *rs1* and *rs2* for the sfence.vma are the same as those used in the sinval.vma,
- reads and writes prior to the sfence.w.inval are considered to be those prior to the sfence.vma, and
- reads and writes following the sfence.inval.ir are considered to be those subsequent to the sfence.vma.

If the hypervisor extension is implemented, the Svinval extension also provides two additional instructions: hinval.vvma and hinval.gvma. These have the same semantics as sinval.vma, except that they combine with sfence.w.inval and sfence.inval.ir to replace hfence.vvma and hfence.gvma, respectively, instead of sfence.vma. In addition, hinval.gvma uses VMIDs instead of ASIDs.

sinval.vma, hinval.vvma, and hinval.gvma require the same permissions and raise the same exceptions as sfence.vma, hfence.vvma, and hfence.gvma, respectively. In particular, an attempt to execute any of these instructions in U-mode always raises an IllegalInstruction exception, and an attempt to execute sinval.vma or hinval.gvma in S-mode or HS-mode when mstatus.TVM=1 also raises an IllegalInstruction exception. An attempt to execute hinval.vvma or hinval.gvma in VS-mode or VU-mode, or to execute sinval.vma in VU-mode, raises a `VirtualInstruction` exception. When hstatus.VTVM=1, an attempt to execute sinval.vma in VS-mode also raises a `VirtualInstruction` exception.

Attempting to execute sfence.w.inval or sfence.inval.ir in U-mode raises an `IllegalInstruction` exception. Doing so in VU-mode raises a `VirtualInstruction` exception. sfence.w.inval and sfence.inval.ir are unaffected by the mstatus.TVM and hstatus.VTVM fields and hence are always permitted in S-mode and VS-mode.

sfence.w.inval and sfence.inval.ir instructions do not need to be trapped when mstatus.TVM=1 or when hstatus.VTVM=1, as they only have ordering effects but no visible side effects. Trapping of the sinval.vma instruction is sufficient to enable emulation of the intended overall TLB maintenance functionality.

In typical usage, software will invalidate a range of virtual addresses in the address-translation caches by executing an sfence.w.inval instruction, executing a series of sinval.vma, hinval.vvma, or hinval.gvma instructions to the addresses (and optionally ASIDs or VMIDs) in question, and then executing an sfence.inval.ir instruction.

> High-performance implementations will be able to pipeline the address-translation cache invalidation operations, and will defer any pipeline stalls or other memory ordering enforcement until an sfence.w.inval, sfence.inval.ir, sfence.vma, hfence.gvma, or hfence.vvma instruction is executed.
>
> Simpler implementations may implement sinval.vma, hinval.vvma, and hinval.gvma identically to sfence.vma, hfence.vvma, and hfence.gvma, respectively, while implementing sfence.w.inval and sfence.inval.ir instructions as no-ops.

B.22.3. Instructions

The following 3 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

sfence.inval.ir	**Order implicit page table reads after invalidation**
sfence.w.inval	**Order writes before sfence**
sinval.vma	**Invalidate cached address translations**

B.23. Extension Svnapot

Long Name: Naturally Aligned Power-of-Two Translation Contiguity
Version Requirement: ~> 1.0
Svnapot Extension Presence

profile	**v1.0.0**
RVA22U64	-
RVA22S64	optional

B.23.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-11

B.23.2. Synopsis

In Sv39, Sv48, and Sv57, when a PTE has N=1, the PTE represents a translation that is part of a range of contiguous virtual-to-physical translations with the same values for PTE bits 5-0. Such ranges must be of a naturally aligned power-of-2 (NAPOT) granularity larger than the base page size.

The Svnapot extension depends on Sv39.

Table 9. Page table entry encodings when pte.N=1

i	*pte.ppn*[*i*]	**Description**	*pte.napot_bits*
0	`x xxxx xxx1`	*Reserved*	-
0	`x xxxx xx1x`	*Reserved*	-
0	`x xxxx x1xx`	*Reserved*	-
0	`x xxxx 1000`	64 KiB contiguous region	4
0	`x xxxx 0xxx`	*Reserved*	-
≥1	`x xxxx xxxx`	*Reserved*	-

NAPOT PTEs behave identically to non-NAPOT PTEs within the address-translation algorithm in [sv32algorithm], except that:

- If the encoding in *pte* is valid according to Table 9, then instead of returning the original value of *pte*, implicit reads of a NAPOT PTE return a copy of *pte* in which *pte.ppn*[*i*][*pte.napot_bits*-1:0] is replaced by *vpn*[*i*][*pte.napot_bits*-1:0]. If the encoding in *pte* is reserved according to Table 9, then a page-fault exception must be raised.
- Implicit reads of NAPOT page table entries may create address-translation cache entries mapping *a* + *j**PTESIZE to a copy of *pte* in which *pte.ppn*[*i*][*pte.napot_bits*-1:0] is replaced by *vpn[i][pte.napot_bits*-1:0], for any or all *j* such that *j* >> *napot_bits* = *vpn*[*i*] >> *napot_bits*, all for the address space identified in *satp* as loaded by step 1.

> The motivation for a NAPOT PTE is that it can be cached in a TLB as one or more entries representing the contiguous region as if it were a single (large) page covered by a single translation. This compaction can help relieve TLB pressure in some scenarios. The encoding is designed to fit within the pre-existing Sv39, Sv48, and Sv57 PTE formats so as not to disrupt existing implementations or designs that choose not to implement the scheme. It is also designed so as not to complicate the definition of the address-translation algorithm.
>
> The address translation cache abstraction captures the behavior that would result from the creation of a single TLB entry covering the entire NAPOT region. It is also designed to be consistent with implementations that support NAPOT PTEs by splitting the NAPOT region into TLB entries covering any smaller power-of-two region sizes. For example, a 64 KiB NAPOT PTE might trigger the creation of 16 standard 4 KiB TLB entries, all with contents generated from the NAPOT PTE (even if the PTEs for the other 4 KiB regions have different contents).

In typical usage scenarios, NAPOT PTEs in the same region will have the same attributes, same PPNs, and same values for bits 5-0. RSW remains reserved for supervisor software control. It is the responsibility of the OS and/or hypervisor to configure the page tables in such a way that there are no inconsistencies between NAPOT PTEs and other NAPOT or non-NAPOT PTEs that overlap the same address range. If an update needs to be made, the OS generally should first mark all of the PTEs invalid, then issue SFENCE.VMA instruction(s) covering all 4 KiB regions within the range (either via a single SFENCE.VMA with rs1=<code>x0</code>, or with multiple SFENCE.VMA instructions with rs1≠`x0`), then update the PTE(s), as described in [sfence.vma], unless any inconsistencies are known to be benign. If any inconsistencies do exist, then the effect is the same as when SFENCE.VMA is used incorrectly: one of the translations will be chosen, but the choice is unpredictable.

If an implementation chooses to use a NAPOT PTE (or cached version thereof), it might not consult the PTE directly specified by the algorithm in [sv32algorithm] at all. Therefore, the D and A bits may not be identical across all mappings of the same address range even in typical use cases The operating system must query all NAPOT aliases of a page to determine whether that page has been accessed and/or is dirty. If the OS manually sets the A and/or D bits for a page, it is recommended that the OS also set the A and/or D bits for other NAPOT aliases as appropriate in order to avoid unnecessary traps.

Just as with normal PTEs, TLBs are permitted to cache NAPOT PTEs whose V (Valid) bit is clear.

Depending on need, the NAPOT scheme may be extended to other intermediate page sizes and/or to other levels of the page table in the future. The encoding is designed to accommodate other NAPOT sizes should that need arise. For example:

—

i	*pte.ppn*[*i*]	Description	*pte.napot_bits*
0	`x xxxx xxx1`	8 KiB contiguous region	1
0	`x xxxx xx10`	16 KiB contiguous region	2
0	`x xxxx x100`	32 KiB contiguous region	3
0	`x xxxx 1000`	64 KiB contiguous region	4
0	`x xxx1 0000`	128 KiB contiguous region	5
…	…	…	…
1	`x xxxx xxx1`	4 MiB contiguous region	1
1	`x xxxx xx10`	8 MiB contiguous region	2
…	…	…	…

In such a case, an implementation may or may not support all options. The discoverability mechanism for this extension would be extended to allow system software to determine which sizes are supported.

Other sizes may remain deliberately excluded, so that PPN bits not being used to indicate a valid NAPOT region size (e.g., the least-significant bit of *pte.ppn*[*i*]) may be repurposed for other uses in the future.

However, in case finer-grained intermediate page size support proves not to be useful, we have chosen to standardize only 64 KiB support as a first step.

It is expected that Svnapot will be mandatory in the next profile release.

B.24. Extension Svpbmt

Long Name: Page-based memory types
Version Requirement: ~> 1.0
Svpbmt Extension Presence

profile	**v1.0.0**
RVA22U64	-
RVA22S64	mandatory

B.24.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-12

B.24.2. Synopsis

This extension mandates that the satp mode Bare must be supported.

B.25. Extension U

Long Name: User-mode privilege level

Version Requirement: ~> 2.0
U Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.25.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2019-12

B.25.2. Synopsis

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

B.25.3. CSRs

The following 1 CSRs are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
mcounteren	Machine Counter Enable	0x306	M

B.25.4. Parameters

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

MCOUNTENABLE_EN

Indicates which counters can be delegated via mcounteren.

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

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

TRAP_ON_ECALL_FROM_U

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

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

UXLEN

Set of XLENs supported in U-mode. When both 32 and 64 are supported, SXLEN can be changed, via mstatus.UXL, between 32 and 64.

U_MODE_ENDIANNESS

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

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

B.26. Extension V

Long Name: Vector Operations
Version Requirement: ~> 1.0
V Extension Presence

profile	v1.0.0
RVA22U64	optional
RVA22S64	optional

B.26.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-11

B.26.2. Synopsis

General support for data-parallel execution.

The smaller vector extensions (Zve32f, Zve32x, Zve64d, Zve64f, Zve64x) are not provided as separately supported profile options. The full V extension is specified as the only supported profile option.

A future profile might mandate V.

B.27. Extension Za128rs

Long Name: Reservation set size of at most 128 bytes
Version Requirement: ~> 1.0
Za128rs Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.27.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.27.2. Synopsis

Reservation sets must be contiguous, naturally aligned, and at most 128 bytes in size.

This extension was ratified as part of the RVA20 profile.

The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

Za128rs is a profile-defined extension introduced with RVA20. The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

B.28. Extension Za64rs

Long Name: Reservation set size of at most 64 bytes
Version Requirement: ~> 1.0
Za64rs Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.28.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.28.2. Synopsis

Reservation sets must be contiguous, naturally aligned, and at most 64 bytes in size.

This extension was ratified as part of the RVA22 profile.

The minimum reservation set size is effectively determined by the size of atomic accesses in the A extension.

Reservation sets are contiguous, naturally aligned, and a maximum of 64 bytes

B.29. Extension Zba

Long Name: Address generation
Version Requirement: ~> 1.0
Zba Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.29.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-06

B.29.2. Synopsis

The Zba instructions can be used to accelerate the generation of addresses that index into arrays of basic types (halfword, word, doubleword) using both unsigned word-sized and XLEN-sized indices: a shifted index is added to a base address.

The shift and add instructions do a left shift of 1, 2, or 3 because these are commonly found in real-world code and because they can be implemented with a minimal amount of additional hardware beyond that of the simple adder. This avoids lengthening the critical path in implementations.

While the shift and add instructions are limited to a maximum left shift of 3, the slli instruction (from the base ISA) can be used to perform similar shifts for indexing into arrays of wider elements. The slli.uw — added in this extension — can be used when the index is to be interpreted as an unsigned word.

B.29.3. Instructions

The following 3 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

sh1add	**Shift left by 1 and add**
sh2add	**Shift left by 2 and add**
sh3add	**Shift left by 3 and add**

B.30. Extension Zbb

Long Name: Basic bit-manipulation
Version Requirement: ~> 1.0
Zbb Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.30.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-06

B.30.2. Synopsis

Basic bit-manipulation

B.30.3. Instructions

The following 17 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

andn	**AND with inverted operand**
orn	**OR with inverted operand**
rev8	**Byte-reverse register (RV64 encoding)**
rol	**Rotate left (Register)**

ror	**Rotate right (Register)**
rori	**Rotate right (Immediate)**
xnor	**Exclusive NOR**
clz	**Count leading zero bits**
cpop	**Count set bits**
ctz	**Count trailing zero bits**
max	**Maximum**
maxu	**Unsigned maximum**
min	**Minimum**
minu	**Unsigned minimum**
orc.b	**Bitware OR-combine, byte granule**
sext.b	**Sign-extend byte**
sext.h	**Sign-extend halfword**

B.31. Extension Zbs

Long Name: Single-bit instructions
Version Requirement: ~> 1.0
Zbs Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.31.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-06
Ratification document	drive.google.com/drive/u/0/folders/1_wqb-rXOVkGa6rqmugN3kwCftWDf1daU

B.31.2. Synopsis

The single-bit instructions provide a mechanism to set, clear, invert, or extract a single bit in a register. The bit is specified by its index

B.31.3. Instructions

The following 8 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

bclr	**Single-Bit clear (Register)**
bclri	**Single-Bit clear (Immediate)**
bext	**Single-Bit extract (Register)**
bexti	**Single-Bit extract (Immediate)**
binv	**Single-Bit invert (Register)**
binvi	**Single-Bit invert (Immediate)**
bset	**Single-Bit set (Register)**
bseti	**Single-Bit set (Immediate)**

B.32. Extension Zca

Long Name: C instructions excluding floating-point loads/stores
Version Requirement: ~> 1.0
Zca Extension Presence

profile	**v1.0.0**
RVA22U64	optional
RVA22S64	optional

B.32.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-04

B.32.2. Synopsis

The Zca extension is added as way to refer to instructions in the C extension that do not include the floating-point loads and stores.

Therefore it excludes all 16-bit floating point loads and stores: c.flw, c.flwsp, c.fsw, c.fswsp, c.fld, c.fldsp, c.fsd, c.fsdsp.

The 'C' extension only includes F/D instructions when D and F are also specified.

B.32.3. Instructions

The following 26 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

c.add	**Add**
c.addi	**Add a sign-extended non-zero immediate**
c.addi16sp	**Add a sign-extended non-zero immediate**
c.addi4spn	**Add a zero-extended non-zero immediate, scaled by 4, to the stack pointer**
c.and	**And**
c.andi	**And immediate**
c.beqz	**Branch if Equal Zero**
c.bnez	**Branch if NOT Equal Zero**
c.ebreak	**Breakpoint exception**
c.j	**Jump**
c.jalr	**Jump and Link Register**
c.jr	**Jump Register**
c.li	**Load the sign-extended 6-bit immediate**
c.lui	**Load Upper Immediate**
c.lw	**Load word**
c.lwsp	**Load word from stack pointer**
c.mv	**Move Register**
c.nop	**Non-operation**
c.or	**Or**
c.slli	**Shift left logical immediate**
c.srai	**Shift right arithmetical immediate**
c.srli	**Shift right logical immediate**
c.sub	**Subtract**
c.sw	**Store word**
c.swsp	**Store word to stack**
c.xor	**Exclusive Or**

B.33. Extension Zcd

Long Name: Compressed double-precision floating-point loads/stores
Version Requirement: ~> 1.0
Zcd Extension Presence

profile	**v1.0.0**
RVA22U64	optional
RVA22S64	optional

B.33.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2023-04

B.33.2. Synopsis

Zcd is the existing set of compressed double precision floating point loads and stores: c.fld, c.fldsp, c.fsd, c.fsdsp.

B.33.3. Instructions

The following 4 instructions are added by extension version 1.0.0 (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**

B.34. Extension Zcf

Long Name: Compressed single-precision floating-point loads/stores
Version Requirement: ~> 1.0
Zcf Extension Presence

profile	**v1.0.0**
RVA22U64	optional
RVA22S64	optional

B.34.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2023-04

B.34.2. Synopsis

Zcf is the existing set of compressed single precision floating point loads and stores (RV32 only): c.flw, c.flwsp, c.fsw, c.fswsp.

B.34.3. Instructions

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

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.35. Extension Zfh

Long Name: Half-precision floating point
Version Requirement: ~> 1.0
Zfh Extension Presence

profile	**v1.0.0**
RVA22U64	optional
RVA22S64	optional

B.35.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2021-11

B.35.2. Synopsis

16-bit half-precision binary floating-point instructions compliant with the IEEE 754-2008 arithmetic standard. The Zfh extension depends on the single-precision floating-point extension, F. The NaN-boxing scheme is extended to allow a half-precision value to be NaN-boxed inside a single-precision value (which may be recursively NaN-boxed inside a double- or quad-precision value when the D or Q extension is present).

A future profile might mandate V.

B.35.3. Instructions

The following 22 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

fadd.h	**No synopsis available**
fclass.h	**No synopsis available**
fcvt.h.w	**Floating-point Convert Word to Half-precision**
fcvt.h.wu	**Floating-point Convert Unsigned Word to Half-precision**
fcvt.w.h	**Floating-point Convert Half-precision to Word**
fcvt.wu.h	**Floating-point Convert Half-precision to Word**
fdiv.h	**No synopsis available**
feq.h	**No synopsis available**
fle.h	**No synopsis available**
flt.h	**No synopsis available**
fmadd.h	**No synopsis available**
fmax.h	**No synopsis available**
fmin.h	**No synopsis available**
fmsub.h	**No synopsis available**
fmul.h	**No synopsis available**
fnmadd.h	**No synopsis available**
fnmsub.h	**No synopsis available**
fsgnj.h	**No synopsis available**
fsgnjn.h	**No synopsis available**
fsgnjx.h	**No synopsis available**
fsqrt.h	**No synopsis available**
fsub.h	**No synopsis available**

B.36. Extension Zfhmin

Long Name: Minimal half-precision floating-point
Version Requirement: ~> 1.0
Zfhmin Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.36.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2021-11

B.36.2. Synopsis

Zfhmin provides minimal support for 16-bit half-precision binary floating-point instructions. The Zfhmin extension is a subset of the Zfh extension, consisting only of data transfer and conversion instructions. Like Zfh, the Zfhmin extension depends on the single-precision floating-point extension, F. The expectation is that Zfhmin software primarily uses the half-precision format for storage, performing most computation in higher precision.

The Zfhmin extension includes the following instructions from the Zfh extension: flh, fsh, fmv.x.h, fmv.h.x, fcvt.s.h, and fcvt.h.s. If the D extension is present, the fcvt.d.h and fcvt.h.d instructions are also included. If the Q extension is present, the fcvt.q.h and fcvt.h.q instructions are additionally included.

Zfhmin does not include the fsgnj.h instruction, because it suffices to instead use the fsgnj.s instruction to move half-precision values between floating-point registers.

Half-precision addition, subtraction, multiplication, division, and square-root operations can be faithfully emulated by converting the half-precision operands to single-precision, performing the operation using single-precision arithmetic, then converting back to half-precision. cite:[roux:hal-01091186] Performing half-precision fused multiply-addition using this method incurs a 1-ulp error on some inputs for the RNE and RMM rounding modes.

Conversion from 8- or 16-bit integers to half-precision can be emulated by first converting to single-precision, then converting to half-precision. Conversion from 32-bit integer can be emulated by first converting to double-precision. If the D extension is not present and a 1-ulp error under RNE or RMM is tolerable, 32-bit integers can be first converted to single-precision instead. The same remark applies to conversions from 64-bit integers without the Q extension.

Zfhmin is a small extension that adds support to load/store and convert IEEE 754 half-precision numbers to and from the IEEE 754 single-precision format. The hardware cost for this extension is low, and mandating the extension avoids adding an option to the profile.

B.36.3. Instructions

The following 6 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

fcvt.h.s	**Convert half-precision float to a single-precision float**
fcvt.s.h	**Convert single-precision float to a half-precision float**
flh	**Half-precision floating-point load**
fmv.h.x	**Half-precision floating-point move from integer**
fmv.x.h	**Move half-precision value from floating-point to integer register**
fsh	**Half-precision floating-point store**

B.37. Extension Zic64b

Long Name: 64-byte cache blocks
Version Requirement: = 1.0
Zic64b Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.37.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03
Ratification document	github.com/riscv/riscv-profiles/releases/tag/v1.0

B.37.2. Synopsis

Cache blocks must be 64 bytes in size, naturally aligned in the address space.

This extension was ratified with the RVA20 profiles.

This is a new extension name for this feature. While the general RISC-V specifications are agnostic to cache block size, selecting a common cache block size simplifies the specification and use of the following cache-block extensions within the application processor profile. Software does not have to query a discovery mechanism and/or provide dynamic dispatch to the appropriate code. We choose 64 bytes at it is effectively an industry standard. Implementations may use longer cache blocks to reduce tag cost provided they use 64-byte sub-blocks to remain compatible. Implementations may use shorter cache blocks provided they sequence cache operations across the multiple cache blocks comprising a 64-byte block to remain compatible.

B.38. Extension Zicbom

Long Name: Cache-block management instructions
Version Requirement: ~> 1.0
Zicbom Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.38.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2022-05

B.38.2. Synopsis

Cache-block management instructions

B.38.3. Instructions

The following 3 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

cbo.clean	**Cache Block Clean**
cbo.flush	**Cache Block Flush**
cbo.inval	**Cache Block Invalidate**

B.38.4. Parameters

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

FORCE_UPGRADE_CBO_INVAL_TO_FLUSH

When true, an implementation prohibits setting menvcfg.CBIE == 11 such that all cbo.inval instructions either trap (when menvcfg.CBIE == '00') or flush (when menvcfg.CBIE == '01').

When false, an implementation allows a true INVAL operation for cbo.inval, and thus supports the setting menvcfg.CBIE == 11.

B.39. Extension Zicbop

Long Name: Cache-block prefetch
Version Requirement: ~> 1.0
Zicbop Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.39.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2022-05

B.39.2. Synopsis

Cache-block prefetch instructions

As with other HINTS, the inclusion of prefetches in the mandatory set of extensions indicates that software should generate these instructions where they are expected to be useful, and hardware is expected to exploit that information.

B.39.3. Instructions

The following 3 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

prefetch.i	**Cache block prefetch for instruction fetch**
prefetch.r	**Cache block prefetch for data read**

prefetch.w	**Cache block prefetch for data write**

B.39.4. Parameters

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

B.40. Extension Zicboz

Long Name: Cache-block zero instruction
Version Requirement: ~> 1.0
Zicboz Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.40.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2022-05

B.40.2. Synopsis

Cache-block zero instruction

B.40.3. Instructions

The following 1 instructions are added by extension version 1.0.0 (the minimum version of this extension that satifies the extension requirement).

cbo.zero	**Cache Block Zero**

B.40.4. Parameters

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

CACHE_BLOCK_SIZE

The observable size of a cache block, in bytes

B.41. Extension Ziccamoa

Long Name: Main memory supports all atomics in A extension
Version Requirement: ~> 1.0
Ziccamoa Extension Presence

profile	**v1.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.41.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.41.2. Synopsis

Main memory regions with both the cacheability and coherence PMAs must support AMOArithmetic.

This extension was ratified as part of the RVA20 profile.

Ziccamo is a profile-defined extension introduced with RVA20.

B.42. Extension Ziccif

Long Name: Main memory supports instruction fetch with atomicity requirement
Version Requirement: ~> 1.0
Ziccif Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.42.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.42.2. Synopsis

Main memory regions with both the cacheability and coherence PMAs must support instruction fetch, and any instruction fetches of naturally aligned power-of-2 sizes up to min(ILEN,XLEN) (i.e., 32 bits for RVA22) are atomic.

This extension was ratified as part of the RVA20 profile.

Ziccif is a profile-defined extension introduced with RVA20. The fetch atomicity requirement facilitates runtime patching of aligned instructions.

B.43. Extension Zicclsm

Long Name: Main memory supports misaligned loads/stores
Version Requirement: ~> 1.0
Zicclsm Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.43.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.43.2. Synopsis

Misaligned loads and stores to main memory regions with both the cacheability and coherence PMAs must be supported.

This extension was ratified as part of the RVA20 profile.

This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

Zicclsm is a profile-defined extension introduced with RVA20. This requires misaligned support for all regular load and store instructions (including scalar and vector) but not AMOs or other specialized forms of memory access. Even though mandated, misaligned loads and stores might execute extremely slowly. Standard software distributions should assume their existence only for correctness, not for performance.

B.44. Extension Ziccrse

Long Name: Main memory supports forward progress on LR/SC sequences
Version Requirement: ~> 1.0
Ziccrse Extension Presence

profile	v1.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.44.1. Available Versions

	Version 1.0.0
State	ratified
Ratification date	2023-03

B.44.2. Synopsis

Main memory regions with both the cacheability and coherence PMAs must support RsrvEventual.

This extension was ratified as part of the RVA20 profile.

Ziccrse is a profile-defined extension introduced with RVA20.

B.45. Extension Zicntr

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

profile	v2.0.0
RVA22U64	mandatory
RVA22S64	mandatory

B.45.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-12

B.45.2. Synopsis

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

B.45.3. CSRs

The following 3 CSRs are added by extension version 2.0.0 (the minimum version of this extension that satifies the extension requirement).

Name	Long Name	Address	Mode
cycle	Cycle counter for RDCYCLE Instruction	0xc00	U
instret	Instructions retired counter for RDINSTRET Instruction	0xc02	U
time	Timer for RDTIME Instruction	0xc01	U

B.45.4. Parameters

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

TIME_CSR_IMPLEMENTED

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

Possible values:

true

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

false

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

B.46. Extension Zicsr

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

profile	**v2.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.46.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-04

B.46.2. Synopsis

Control and status register instructions

B.46.3. Instructions

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

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

B.47. Extension Zifencei

Long Name: Instruction fence
Version Requirement: ~> 2.0
Zifencei Extension Presence

profile	**v2.0.0**
RVA22U64	-
RVA22S64	mandatory

B.47.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2019-04

B.47.2. 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.

Zifencei is mandated as it is the only standard way to support instruction-cache coherence in RVA20 application processors. A new instruction-cache coherence mechanism is under development which might be added as an option in the future.

B.47.3. 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.48. Extension Zihintpause

Long Name: Pause Hint
Version Requirement: ~> 2.0
Zihintpause Extension Presence

profile	**v2.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.48.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2021-02

B.48.2. Synopsis

The PAUSE instruction is a HINT that indicates the current hart's rate of instruction retirement should be temporarily reduced or paused. The duration of its effect must be bounded and may be zero.

Software can use the PAUSE instruction to reduce energy consumption while executing spin-wait code sequences. Multithreaded cores might temporarily relinquish execution resources to other harts when PAUSE is executed. It is recommended that a PAUSE instruction generally be included in the code sequence for a spin-wait loop.

A future extension might add primitives similar to the x86 MONITOR/MWAIT instructions, which provide a more efficient mechanism to wait on writes to a specific memory location. However, these instructions would not supplant PAUSE. PAUSE is more appropriate when polling for non-memory events, when polling for multiple events, or when software does not know precisely what events it is polling for.

The duration of a PAUSE instruction's effect may vary significantly within and among implementations. In typical implementations this duration should be much less than the time to perform a context switch, probably more on the rough order of an on-chip cache miss latency or a cacheless access to main memory.

A series of PAUSE instructions can be used to create a cumulative delay loosely proportional to the number of PAUSE instructions. In spin-wait loops in portable code, however, only one PAUSE instruction should be used before re-evaluating loop conditions, else the hart might stall longer than optimal on some implementations, degrading system performance.

PAUSE is encoded as a FENCE instruction with *pred*=`W`, *succ*=`0`, *fm*=`0`, *rd*=`x0`, and *rs1*=`x0`.

PAUSE is encoded as a hint within the FENCE opcode because some implementations are expected to deliberately stall the PAUSE instruction until outstanding memory transactions have completed. Because the successor set is null, however, PAUSE does not *mandate* any particular memory ordering—hence, it truly is a HINT.

Like other FENCE instructions, PAUSE cannot be used within LR/SC sequences without voiding the forward-progress guarantee.

The choice of a predecessor set of W is arbitrary, since the successor set is null. Other HINTs similar to PAUSE might be encoded with other predecessor sets.

While the pause instruction is a HINT can be implemented as a NOP and hence trivially supported by hardware implementers, its inclusion in the mandatory extension list signifies that software should use the instruction whenever it would make sense and that implementers are expected to exploit this information to optimize hardware execution.

B.48.3. Instructions

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

pause	**Pause hint**

B.49. Extension Zihpm

Long Name: Hardware Performance Counters
Version Requirement: ~> 2.0
Zihpm Extension Presence

profile	**v2.0.0**
RVA22U64	mandatory
RVA22S64	mandatory

B.49.1. Available Versions

	Version 2.0.0
State	ratified
Ratification date	2023-03

B.49.2. Synopsis

Hardware performance counters

The number of counters is platform-specific.

B.49.3. 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

Name	Long Name	Address	Mode
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

B.50. Extension Zkn

Long Name: NIST Algorithm Suite
Version Requirement: ~> 1.0
Zkn Extension Presence

profile	v1.0.0	v1.0.1
RVA22U64	optional	optional
RVA22S64	optional	optional

B.50.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2021-12

Version 1.0.1	
State	ratified
Ratification date	2022-02

B.50.2. Synopsis

This extension is shorthand for the following set of other extensions:

- Zbkb
- Zbkc
- Zbkx
- Zkne
- Zknd
- Zknh

B.51. Extension Zkr

Long Name: Entropy Source
Version Requirement: ~> 1.0
Zkr Extension Presence

profile	v1.0.0	v1.0.1
RVA22U64	-	-
RVA22S64	optional	optional

B.51.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2021-12

Version 1.0.1	
State	ratified
Ratification date	2022-02

B.51.2. Synopsis

Defines the `seed` CSR. This CSR provides up to 16 physical entropy bits that can be used to seed cryptographic random bit generators.

Technically, Zk is also a privileged-mode option capturing that Zkr, Zkn, and Zkt are all implemented. However, the Zk rollup is less descriptive than specifying the individual extensions explicitly.

B.52. Extension Zks

Long Name: ShangMi Algorithm Suite
Version Requirement: ~> 1.0
Zks Extension Presence

profile	v1.0.0	v1.0.1
RVA22U64	optional	optional
RVA22S64	optional	optional

B.52.1. Available Versions

Version 1.0.0	
State	ratified
Ratification date	2021-12

Version 1.0.1	
State	ratified
Ratification date	2022-02

B.52.2. Synopsis

This extension is shorthand for the following set of other extensions:

- Zbkb
- Zbkc
- Zbkx
- Zksed
- Zksh

B.53. Extension Zkt

Long Name: Data-independent execution latency
Version Requirement: ~> 1.0
Zkt Extension Presence

profile	v1.0.0	v1.0.1
RVA22U64	mandatory	mandatory
RVA22S64	mandatory	mandatory

B.53.1. Available Versions

Version 1.0.0	
State	ratified

Version 1.0.0	
Ratification date	2021-12

Version 1.0.1	
State	ratified
Ratification date	2022-02
Changes	• Fix typos to show that c.srli, c.srai, and c.slli are Zkt instructions in RV64.

B.53.2. Synopsis

The Zkt extension attests that the machine has data-independent execution time for a safe subset of instructions. This property is commonly called "constant-time" although should not be taken with that literal meaning.

All currently proposed cryptographic instructions (scalar `K` extension) are on this list, together with a set of relevant supporting instructions from I, M, C, and B extensions.

Failure to prevent leakage of sensitive parameters via the direct timing channel is considered a serious security vulnerability and will typically result in a CERT CVE security advisory.

Scope and Goal

An "ISA contract" is made between a programmer and the RISC-V implementation that Zkt instructions do not leak information about processed secret data (plaintext, keying information, or other "sensitive security parameters" — FIPS 140-3 term) through differences in execution latency. Zkt does *not* define a set of instructions available in the core; it just restricts the behaviour of certain instructions if those are implemented.

Currently, the scope of this document is within scalar RV32/RV64 processors. Vector cryptography instructions (and appropriate vector support instructions) will be added later, as will other security-related functions that wish to assert leakage-free execution latency properties.

Loads, stores, conditional branches are excluded, along with a set of instructions that are rarely necessary to process secret data. Also excluded are instructions for which workarounds exist in standard cryptographic middleware due to the limitations of other ISA processors.

The stated goal is that OpenSSL, BoringSSL (Android), the Linux Kernel, and similar trusted software will not have directly observable timing side channels when compiled and running on a Zkt-enabled RISC-V target. The Zkt extension explicitly states many of the common latency assumptions made by cryptography developers.

Vendors do not have to implement all of the list's instructions to be Zkt compliant; however, if they claim to have Zkt and implement any of the listed instructions, it must have data-independent latency.

For example, many simple RV32I and RV64I cores (without Multiply, Compressed, Bitmanip, or Cryptographic extensions) are technically compliant with Zkt. A constant-time AES can be implemented on them using "bit-slice" techniques, but it will be excruciatingly slow when compared to implementation with AES instructions. There are no guarantees that even a bit-sliced cipher implementation (largely based on boolean logic instructions) is secure on a core without Zkt attestation.

Out-of-order implementations adhering to Zkt are still free to fuse, crack, change or even ignore sequences of instructions, so long as the optimisations are applied deterministically, and not based on operand data. The guiding principle should be that no information about the data being operated on should be leaked based on the execution latency.

It is left to future extensions or other techniques to tackle the problem of data-independent execution in implementations which advanced out-of-order capabilities which use value prediction, or which are otherwise data-dependent.

Note to software developers

Programming techniques can only mitigate leakage directly caused by arithmetic, caches, and branches. Other ISAs have had micro-architectural issues such as Spectre, Meltdown, Speculative Store Bypass, Rogue System Register Read, Lazy FP State Restore, Bounds Check Bypass Store, TLBleed, and L1TF/Foreshadow, etc. See e.g. NSA Hardware and Firmware Security Guidance

It is not within the remit of this proposal to mitigate these *micro-architectural* leakages.

Background

- Timing attacks are much more powerful than was realised before the 2010s, which has led to a significant mitigation effort in current cryptographic code-bases.
- Cryptography developers use static and dynamic security testing tools to trace the handling of secret information and detect occasions where it influences a branch or is used for a table lookup.
- Architectural testing for Zkt can be pragmatic and semi-formal; *security by design* against basic timing attacks can usually be achieved via conscious implementation (of relevant iterative multi-cycle instructions or instructions composed of micro-ops) in way that avoids data-dependent latency.
- Laboratory testing may utilize statistical timing attack leakage analysis techniques such as those described in ISO/IEC 17825 cite:[IS16].
- Binary executables should not contain secrets in the instruction encodings (Kerckhoffs's principle), so instruction timing may leak information

about immediates, ordering of input registers, etc. There may be an exception to this in systems where a binary loader modifies the executable for purposes of relocation — and it is desirable to keep the execution location (PC) secret. This is why instructions such as LUI, AUIPC, and ADDI are on the list.

- The rules used by audit tools are relatively simple to understand. Very briefly; we call the plaintext, secret keys, expanded keys, nonces, and other such variables "secrets". A secret variable (arithmetically) modifying any other variable/register turns that into a secret too. If a secret ends up in address calculation affecting a load or store, that is a violation. If a secret affects a branch's condition, that is also a violation. A secret variable location or register becomes a non-secret via specific zeroization/sanitisation or by being declared ciphertext (or otherwise no-longer-secret information). In essence, secrets can only "touch" instructions on the Zkt list while they are secrets.

Specific Instruction Rationale

- HINT instruction forms (typically encodings with `rd=x0`) are excluded from the data-independent time requirement.
- Floating point (F, D, Q, L extensions) are currently excluded from the constant-time requirement as they have very few applications in standardised cryptography. We may consider adding floating point add, sub, multiply as a constant time requirement for some floating point extension in case a specific algorithm (such as the PQC Signature algorithm Falcon) becomes critical.
- Cryptographers typically assume division to be variable-time (while multiplication is constant time) and implement their Montgomery reduction routines with that assumption.
- Zicsr, Zifencei are excluded.
- Some instructions are on the list simply because we see no harm in including them in testing scope.

Programming Information

For background information on secure programming "models", see:

- Thomas Pornin: *"Why Constant-Time Crypto?"* (A great introduction to timing assumptions.) www.bearssl.org/constanttime.html
- Jean-Philippe Aumasson: *"Guidelines for low-level cryptography software."* (A list of recommendations.) github.com/veorq/cryptocoding
- Peter Schwabe: *"Timing Attacks and Countermeasures."* (Lecture slides — nice references.) summerschool-croatia.cs.ru.nl/2016/slides/PeterSchwabe.pdf
- Adam Langley: *"ctgrind."* (This is from 2010 but is still relevant.) www.imperialviolet.org/2010/04/01/ctgrind.html
- Kris Kwiatkowski: *"Constant-time code verification with Memory Sanitizer."* www.amongbytes.com/post/20210709-testing-constant-time/
- For early examples of timing attack vulnerabilities, see www.kb.cert.org/vuls/id/997481 and related academic papers.

Zkt listings

The following instructions are included in the Zkt subset They are listed here grouped by their original parent extension.

Note to implementers

You do not need to implement all of these instructions to implement Zkt. Rather, every one of these instructions that the core does implement must adhere to the requirements of Zkt.

RVI (Base Instruction Set)

Only basic arithmetic and `slt*` (for carry computations) are included. The data-independent timing requirement does not apply to HINT instruction encoding forms of these instructions.

RV32	RV64	Mnemonic	Instruction
✔	✔	lui *rd, imm*	'lui'
✔	✔	auipc *rd, imm*	'auipc'
✔	✔	addi *rd, rs1, imm*	'addi'
✔	✔	slti *rd, rs1, imm*	'slti'
✔	✔	sltiu *rd, rs1, imm*	'sltiu'
✔	✔	xori *rd, rs1, imm*	'xori'
✔	✔	ori *rd, rs1, imm*	'ori'
✔	✔	andi *rd, rs1, imm*	'andi'
✔	✔	slli *rd, rs1, imm*	'slli'
✔	✔	srli *rd, rs1, imm*	'srli'
✔	✔	srai *rd, rs1, imm*	'srai'
✔	✔	add *rd, rs1, rs2*	'add'
✔	✔	sub *rd, rs1, rs2*	'sub'
✔	✔	sll *rd, rs1, rs2*	'sll'
✔	✔	slt *rd, rs1, rs2*	'slt'

RV32	RV64	Mnemonic	Instruction
✔	✔	sltu *rd*, *rs1*, *rs2*	'sltu'
✔	✔	xor *rd*, *rs1*, *rs2*	'xor'
✔	✔	srl *rd*, *rs1*, *rs2*	'srl'
✔	✔	sra *rd*, *rs1*, *rs2*	'sra'
✔	✔	or *rd*, *rs1*, *rs2*	'or'
✔	✔	and *rd*, *rs1*, *rs2*	'and'
	✔	addiw *rd*, *rs1*, *imm*	'addiw'
	✔	slliw *rd*, *rs1*, *imm*	'slliw'
	✔	srliw *rd*, *rs1*, *imm*	'srliw'
	✔	sraiw *rd*, *rs1*, *imm*	'sraiw'
	✔	addw *rd*, *rs1*, *rs2*	'addw'
	✔	subw *rd*, *rs1*, *rs2*	'subw'
	✔	sllw *rd*, *rs1*, *rs2*	'sllw'
	✔	srlw *rd*, *rs1*, *rs2*	'srlw'
	✔	sraw *rd*, *rs1*, *rs2*	'sraw'

RVM (Multiply)

Multiplication is included; division and remaindering excluded.

RV32	RV64	Mnemonic	Instruction
✔	✔	mul *rd*, *rs1*, *rs2*	'mul'
✔	✔	mulh *rd*, *rs1*, *rs2*	'mulh'
✔	✔	mulhsu *rd*, *rs1*, *rs2*	'mulhsu'
✔	✔	mulhu *rd*, *rs1*, *rs2*	'mulhu'
	✔	mulw *rd*, *rs1*, *rs2*	'mulw'

RVC (Compressed)

Same criteria as in RVI. Organised by quadrants.

RV32	RV64	Mnemonic	Instruction
✔	✔	c.nop	'c_nop'
✔	✔	c.addi	'c_addi'
	✔	c.addiw	'c_addiw'
✔	✔	c.lui	'c_lui'
✔	✔	c.srli	'c_srli'
✔	✔	c.srai	'c_srai'
✔	✔	c.andi	'c_andi'
✔	✔	c.sub	'c_sub'
✔	✔	c.xor	'c_xor'
✔	✔	c.or	'c_or'
✔	✔	c.and	'c_and'
	✔	c.subw	'c_subw'
	✔	c.addw	'c_addw'
✔	✔	c.slli	'c_slli'
✔	✔	c.mv	'c_mv'
✔	✔	c.add	'c_add'

RVK (Scalar Cryptography)

All K-specific instructions are included. Additionally, `seed` CSR latency should be independent of `ES16` state output `entropy` bits, as that is a sensitive security parameter. See <<crypto_scalar_appx_es_access'.

RV32	RV64	Mnemonic	Instruction
✔		aes32dsi	'aes32dsi'
✔		aes32dsmi	'aes32dsmi'
✔		aes32esi	'aes32esi'
✔		aes32esmi	'aes32esmi'
	✔	aes64ds	'aes64ds'
	✔	aes64dsm	'aes64dsm'
	✔	aes64es	'aes64es'
	✔	aes64esm	'aes64esm'
	✔	aes64im	'aes64im'
	✔	aes64ks1i	'aes64ks1i'
	✔	aes64ks2	'aes64ks2'
✔	✔	sha256sig0	'sha256sig0'
✔	✔	sha256sig1	'sha256sig1'
✔	✔	sha256sum0	'sha256sum0'
✔	✔	sha256sum1	'sha256sum1'
✔		sha512sig0h	'sha512sig0h'
✔		sha512sig0l	'sha512sig0l'
✔		sha512sig1h	'sha512sig1h'
✔		sha512sig1l	'sha512sig1l'
✔		sha512sum0r	'sha512sum0r'
✔		sha512sum1r	'sha512sum1r'
	✔	sha512sig0	'sha512sig0'
	✔	sha512sig1	'sha512sig1'
	✔	sha512sum0	'sha512sum0'
	✔	sha512sum1	'sha512sum1'
✔	✔	sm3p0	'sm3p0'
✔	✔	sm3p1	'sm3p1'
✔	✔	sm4ed	'sm4ed'
✔	✔	sm4ks	'sm4ks'

RVB (Bitmanip)

The Zbkb, `Zbkc' and `Zbkx' extensions are included in their entirety.

Note to implementers

Recall that `rev`, zip and unzip are pseudoinstructions representing specific instances of `grevi`, `shfli` and `unshfli` respectively.

RV32	RV64	Mnemonic	Instruction
✔	✔	clmul	'clmul-sc'
✔	✔	clmulh	'clmulh-sc'
✔	✔	xperm4	'xperm4-sc'
✔	✔	xperm8	'xperm8-sc'
✔	✔	ror	'ror-sc'
✔	✔	rol	'rol-sc'
✔	✔	rori	'rori-sc'
	✔	rorw	'rorw-sc'
	✔	rolw	'rolw-sc'
	✔	roriw	'roriw-sc'
✔	✔	andn	'andn-sc'
✔	✔	orn	'orn-sc'
✔	✔	xnor	'xnor-sc'
✔	✔	pack	'pack-sc'

RV32	RV64	Mnemonic	Instruction
✔	✔	packh	'packh-sc'
	✔	packw	'packw-sc'
✔	✔	brev8	'brev8-sc'
✔	✔	rev8	'rev8-sc'
✔		zip	'zip-sc'
✔		unzip	'unzip-sc'

Zkt requires a certain subset of integer instructions execute with data-independent latency. Mandating this feature enables portable libraries for safe basic cryptographic operations. It is expected that application processors will naturally have this property and so implementation cost is low, if not zero, in most systems that would support RVA22.

Appendix C: Instruction Details

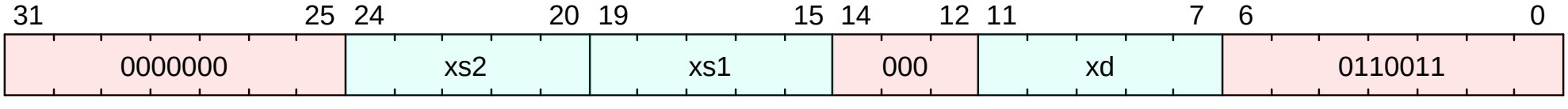
C.1. add

Integer add

This instruction is defined by:

I

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.

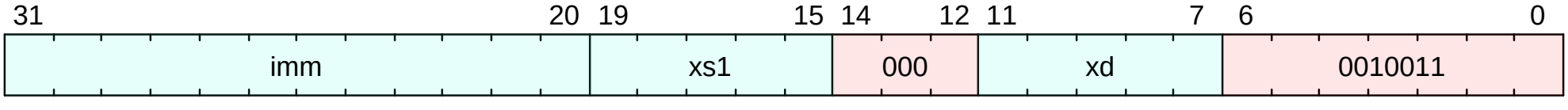
C.2. addi

Add immediate

This instruction is defined by:

I

C.2.1. Encoding



C.2.2. Description

Adds an immediate value 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.

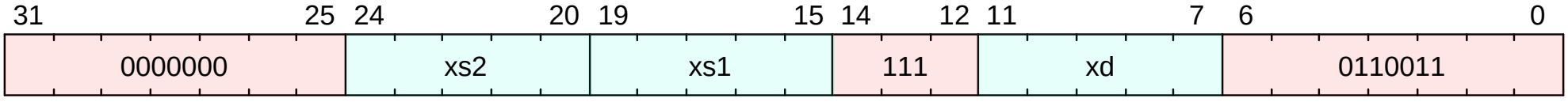
C.3. and

And

This instruction is defined by:

I

C.3.1. Encoding



C.3.2. Description

And xs1 with xs2, and store the result in xd

C.3.3. Access

M	S	U
Always	Always	Always

C.3.4. Decode Variables

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

C.3.5. IDL Operation

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

C.3.6. Sail Operation

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

C.3.7. Exceptions

This instruction does not generate synchronous exceptions.

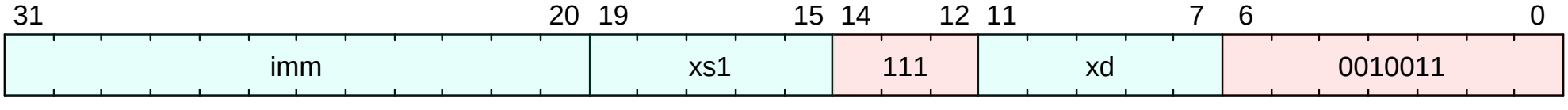
C.4. andi

And immediate

This instruction is defined by:

I

C.4.1. Encoding



C.4.2. Description

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

C.4.3. Access

M	S	U
Always	Always	Always

C.4.4. Decode Variables

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

C.4.5. IDL Operation

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

C.4.6. Sail Operation

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

C.4.7. Exceptions

This instruction does not generate synchronous exceptions.

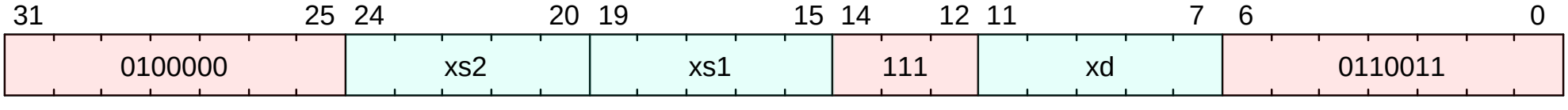
C.5. andn

AND with inverted operand

This instruction is defined by:

(Zbb || Zbkb)

C.5.1. Encoding



C.5.2. Description

Performs the bitwise logical AND operation between `xs1` and the bitwise inversion of `xs2`.

C.5.3. Access

M	S	U
Always	Always	Always

C.5.4. Decode Variables

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

C.5.5. IDL Operation

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

C.5.6. Sail Operation

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

C.5.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

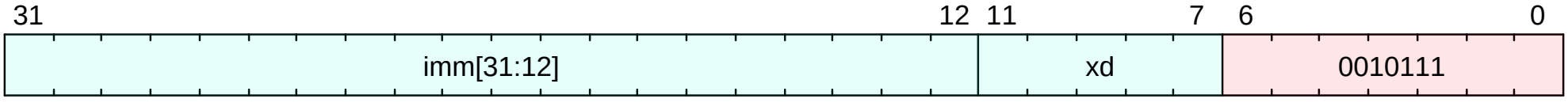
C.6. auipc

Add upper immediate to pc

This instruction is defined by:

I

C.6.1. Encoding



C.6.2. Description

Add an immediate to the current PC.

C.6.3. Access

M	S	U
Always	Always	Always

C.6.4. Decode Variables

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

C.6.5. IDL Operation

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

C.6.6. 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.6.7. Exceptions

This instruction does not generate synchronous exceptions.

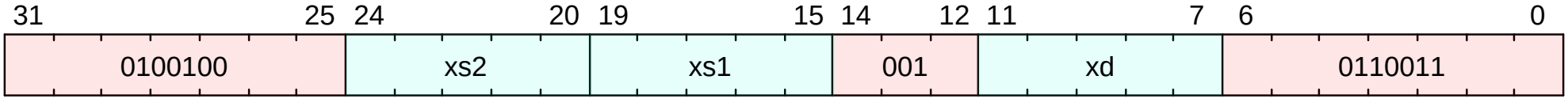
C.7. bclr

Single-Bit clear (Register)

This instruction is defined by:

Zbs

C.7.1. Encoding



C.7.2. Description

Returns xs1 with a single bit cleared at the index specified in xs2. The index is read from the lower log2(XLEN) bits of xs2.

C.7.3. Access

M	S	U
Always	Always	Always

C.7.4. Decode Variables

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

C.7.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[xs2] & (xlen() - 1);
X[xd] = X[xs1] & ~(1 << index);
```

C.7.6. Sail Operation

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

C.7.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.8. bclri

Single-Bit clear (Immediate)

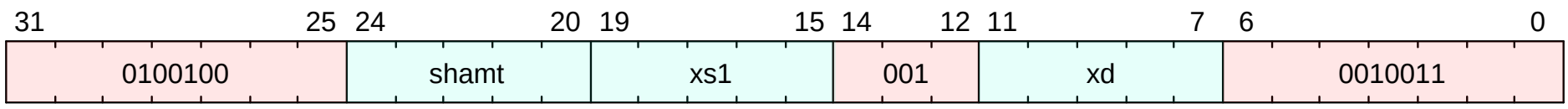
This instruction is defined by:

Zbs

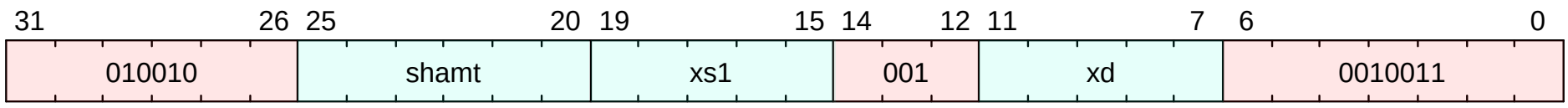
C.8.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.8.2. Description

Returns xs1 with a single bit cleared at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.8.3. Access

M	S	U
Always	Always	Always

C.8.4. 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.8.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[xd] = X[xs1] & ~(1 << index);
```

C.8.6. Sail Operation

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

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

C.8.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

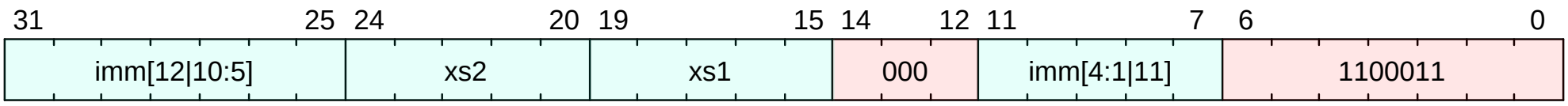
C.9. beq

Branch if equal

This instruction is defined by:

I

C.9.1. Encoding



C.9.2. 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.9.3. Access

M	S	U
Always	Always	Always

C.9.4. Decode Variables

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

C.9.5. IDL Operation

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

C.9.6. Sail Operation

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

```
    }
  } else RETIRE_SUCCESS
}
```

C.9.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

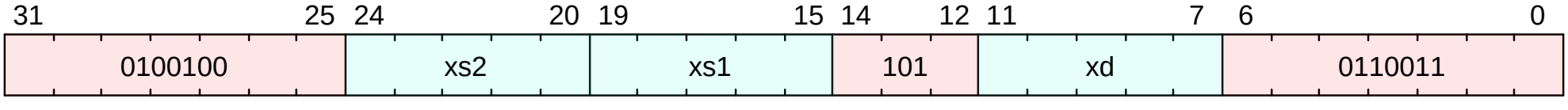
C.10. bext

Single-Bit extract (Register)

This instruction is defined by:

Zbs

C.10.1. Encoding



C.10.2. Description

Returns a single bit extracted from xs1 at the index specified in xs2. The index is read from the lower log2(XLEN) bits of xs2.

C.10.3. Access

M	S	U
Always	Always	Always

C.10.4. Decode Variables

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

C.10.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[xs2] & (xlen() - 1);
X[xd] = (X[xs1] >> index) & 1;
```

C.10.6. Sail Operation

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

C.10.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.11. bexti

Single-Bit extract (Immediate)

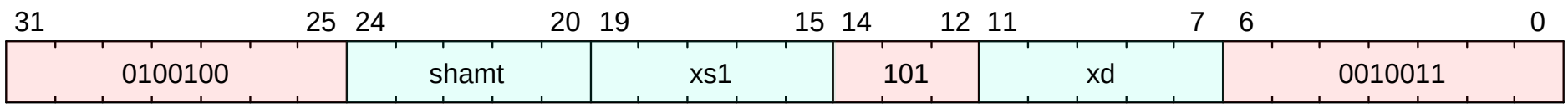
This instruction is defined by:

Zbs

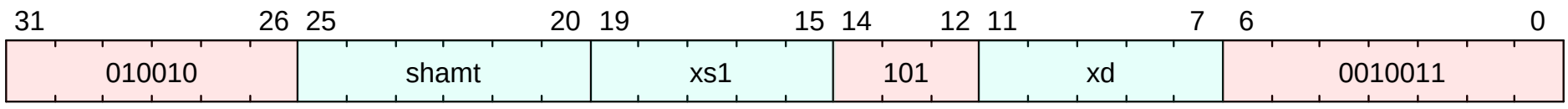
C.11.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.11.2. Description

Returns a single bit extracted from xs1 at the index specified in xs2. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.11.3. Access

M	S	U
Always	Always	Always

C.11.4. 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.11.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[xd] = (X[xs1] >> index) & 1;
```

C.11.6. Sail Operation

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

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

C.11.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

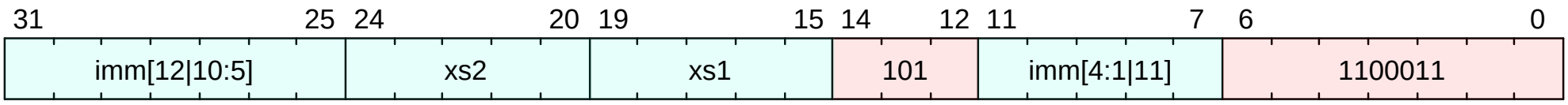
C.12. bge

Branch if greater than or equal

This instruction is defined by:

I

C.12.1. Encoding



C.12.2. 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.12.3. Access

M	S	U
Always	Always	Always

C.12.4. Decode Variables

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

C.12.5. IDL Operation

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

C.12.6. Sail Operation

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

```
    }
  } else RETIRE_SUCCESS
}
```

C.12.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

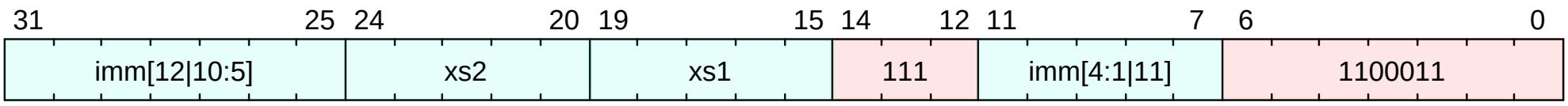
C.13. bgeu

Branch if greater than or equal unsigned

This instruction is defined by:

I

C.13.1. Encoding



C.13.2. 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.13.3. Access

M	S	U
Always	Always	Always

C.13.4. 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.13.5. IDL Operation

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

C.13.6. 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.13.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

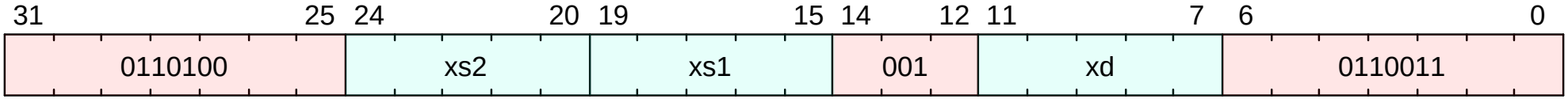
C.14. binv

Single-Bit invert (Register)

This instruction is defined by:

Zbs

C.14.1. Encoding



C.14.2. Description

Returns xs1 with a single bit inverted at the index specified in xs2. The index is read from the lower log2(XLEN) bits of xs2.

C.14.3. Access

M	S	U
Always	Always	Always

C.14.4. Decode Variables

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

C.14.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[xs2] & (xlen() - 1);
X[xd] = X[xs1] ^ (1 << index);
```

C.14.6. Sail Operation

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

C.14.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.15. binvi

Single-Bit invert (Immediate)

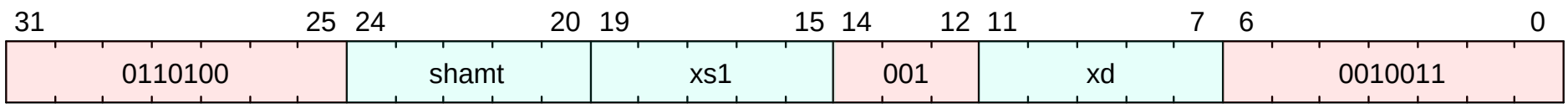
This instruction is defined by:

Zbs

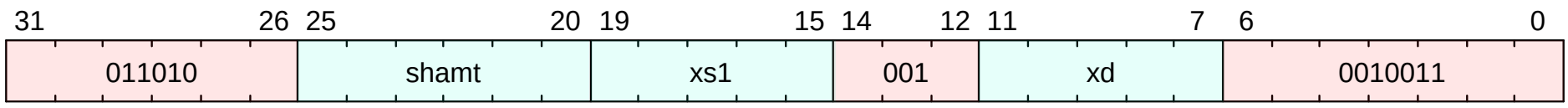
C.15.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.15.2. Description

Returns xs1 with a single bit inverted at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.15.3. Access

M	S	U
Always	Always	Always

C.15.4. 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.15.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[xd] = X[xs1] ^ (1 << index);
```

C.15.6. Sail Operation

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

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

C.15.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

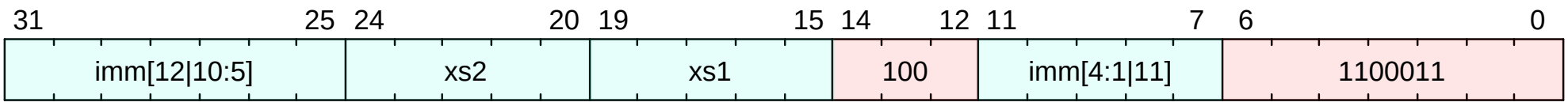
C.16. blt

Branch if less than

This instruction is defined by:

I

C.16.1. Encoding



C.16.2. 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.16.3. Access

M	S	U
Always	Always	Always

C.16.4. Decode Variables

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

C.16.5. IDL Operation

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

C.16.6. Sail Operation

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

```
    }
  } else RETIRE_SUCCESS
}
```

C.16.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

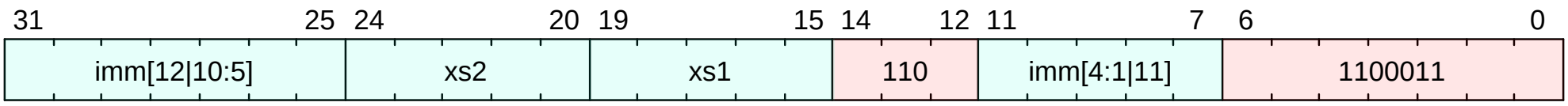
C.17. bltu

Branch if less than unsigned

This instruction is defined by:

I

C.17.1. Encoding



C.17.2. 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.17.3. Access

M	S	U
Always	Always	Always

C.17.4. 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.17.5. IDL Operation

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

C.17.6. 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.17.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

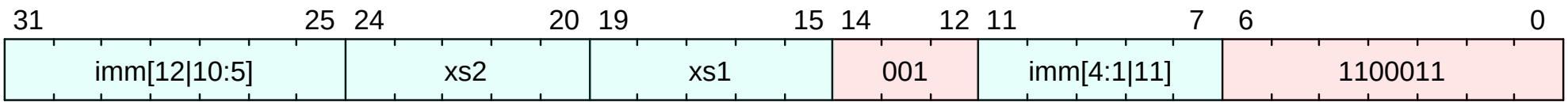
C.18. bne

Branch if not equal

This instruction is defined by:

I

C.18.1. Encoding



C.18.2. 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.18.3. Access

M	S	U
Always	Always	Always

C.18.4. Decode Variables

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

C.18.5. IDL Operation

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

C.18.6. 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.18.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

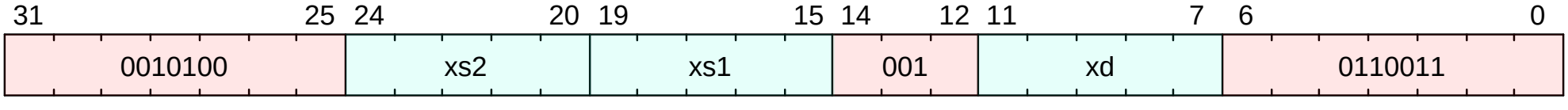
C.19. bset

Single-Bit set (Register)

This instruction is defined by:

Zbs

C.19.1. Encoding



C.19.2. Description

Returns xs1 with a single bit set at the index specified in xs2. The index is read from the lower log2(XLEN) bits of xs2.

C.19.3. Access

M	S	U
Always	Always	Always

C.19.4. Decode Variables

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

C.19.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = X[xs2] & (xlen() - 1);
X[xd] = X[xs1] | (1 << index);
```

C.19.6. Sail Operation

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

C.19.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.20. bseti

Single-Bit set (Immediate)

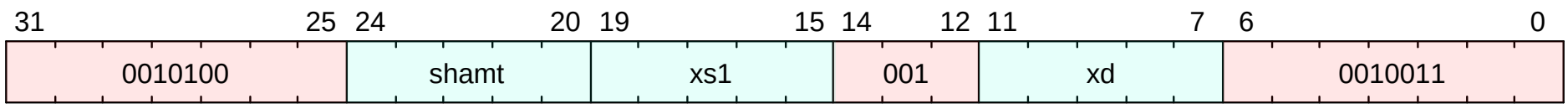
This instruction is defined by:

Zbs

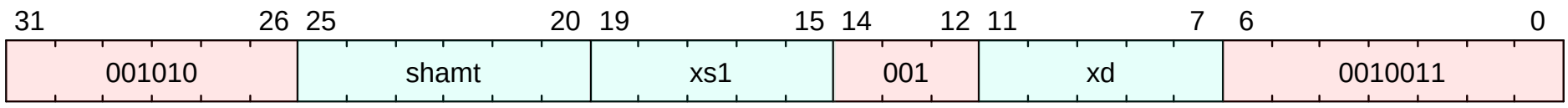
C.20.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.20.2. Description

Returns xs1 with a single bit set at the index specified in shamt. The index is read from the lower log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.20.3. Access

M	S	U
Always	Always	Always

C.20.4. 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.20.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg index = shamt & (xlen() - 1);
X[xd] = X[xs1] | (1 << index);
```

C.20.6. Sail Operation

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

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

C.20.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

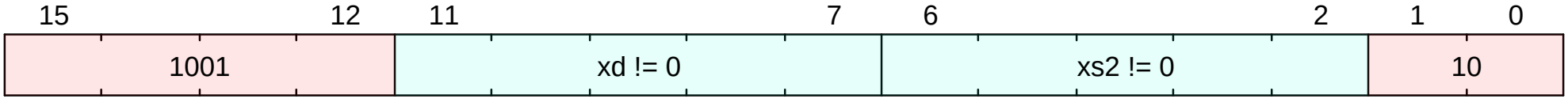
C.21. c.add

Add

This instruction is defined by:

Zca

C.21.1. Encoding



C.21.2. Description

Add the value in xs2 to xd, and store the result in xd. C.ADD expands into `add xd, xd, xs2`.

C.21.3. Access

M	S	U
Always	Always	Always

C.21.4. Decode Variables

```
Bits<5> xs2 = $encoding[6:2];
Bits<5> xd = $encoding[11:7];
```

C.21.5. IDL Operation

```
XReg t0 = X[xd];
XReg t1 = X[xs2];
X[xd] = t0 + t1;
```

C.21.6. Sail Operation

```
{
  let rs1_val = X(rd);
  let rs2_val = X(rs2);
  X(rd) = rs1_val + rs2_val;
  RETIRE_SUCCESS
}
```

C.21.7. Exceptions

This instruction does not generate synchronous exceptions.

C.22. c.addi

Add a sign-extended non-zero immediate

This instruction is defined by:

Zca

C.22.1. Encoding

C.22.2. Description

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

C.22.3. Access

M	S	U
Always	Always	Always

C.22.4. Decode Variables

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

C.22.5. IDL Operation

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

C.22.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

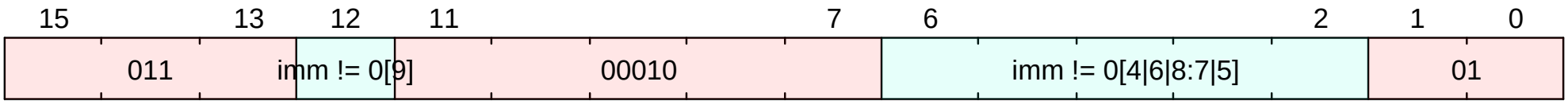
C.23. c.addi16sp

Add a sign-extended non-zero immediate

This instruction is defined by:

Zca

C.23.1. Encoding



C.23.2. Description

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

C.23.3. Access

M	S	U
Always	Always	Always

C.23.4. Decode Variables

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

C.23.5. IDL Operation

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

C.23.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

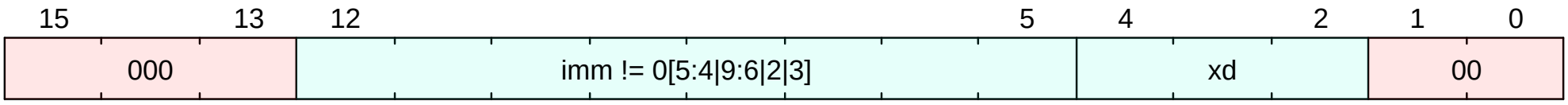
C.24. c.addi4spn

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

This instruction is defined by:

Zca

C.24.1. Encoding



C.24.2. Description

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

C.24.3. Access

M	S	U
Always	Always	Always

C.24.4. Decode Variables

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

C.24.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[creg2reg(xd)] = X[2] + imm;
```

C.24.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

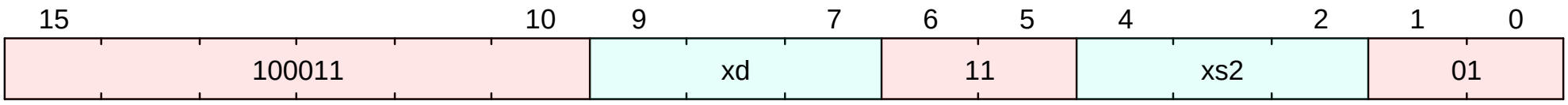
C.25. c.and

And

This instruction is defined by:

Zca

C.25.1. Encoding



C.25.2. Description

And xd with xs2, and store the result in xd The xd and xs2 register indexes should be used as xd+8 and xs2+8 (registers x8-x15). C.AND expands into `and xd, xd, xs2`.

C.25.3. Access

M	S	U
Always	Always	Always

C.25.4. Decode Variables

```
Bits<3> xs2 = $encoding[4:2];
Bits<3> xd = $encoding[9:7];
```

C.25.5. IDL Operation

```
XReg t0 = X[creg2reg(xd)];
XReg t1 = X[creg2reg(xs2)];
X[creg2reg(xd)] = t0 & t1;
```

C.25.6. Sail Operation

```
{
  let rs1_val = X(rd+8);
  let rs2_val = X(rs2+8);
  let result : xlenbits = match op {
    RISCV_ADD  => rs1_val + rs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(rs1_val <_s rs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(rs1_val <_u rs2_val)),
    RISCV_AND  => rs1_val & rs2_val,
    RISCV_OR   => rs1_val | rs2_val,
    RISCV_XOR  => rs1_val ^ rs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then rs1_val << (rs2_val[4..0])
                  else rs1_val << (rs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then rs1_val >> (rs2_val[4..0])
                  else rs1_val >> (rs2_val[5..0]),
    RISCV_SUB  => rs1_val - rs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(rs1_val, rs2_val[4..0])
                  else shift_right_arith64(rs1_val, rs2_val[5..0])
  };
  X(rd+8) = result;
  RETIRE_SUCCESS
}
```

C.25.7. Exceptions

This instruction does not generate synchronous exceptions.

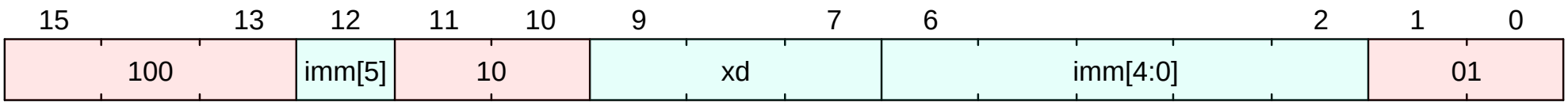
C.26. c.andi

And immediate

This instruction is defined by:

Zca

C.26.1. Encoding



C.26.2. Description

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

C.26.3. Access

M	S	U
Always	Always	Always

C.26.4. Decode Variables

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

C.26.5. IDL Operation

```
X[creg2reg(xd)] = X[creg2reg(xd)] & $signed(imm);
```

C.26.6. Sail Operation

```
{
  let rd_val = X(rd+8);
  let immext : xlenbits = sign_extend(imm);
  let result : xlenbits = match op {
    RISCV_ADDI  => rd_val + immext,
    RISCV_SLTI  => zero_extend(bool_to_bits(rd_val <_s immext)),
    RISCV_SLTIU => zero_extend(bool_to_bits(rd_val <_u immext)),
    RISCV_ANDI  => rd_val & immext,
    RISCV_ORI   => rd_val | immext,
    RISCV_XORI  => rd_val ^ immext
  };
  X(rd+8) = result;
  RETIRE_SUCCESS
}
```

C.26.7. Exceptions

This instruction does not generate synchronous exceptions.

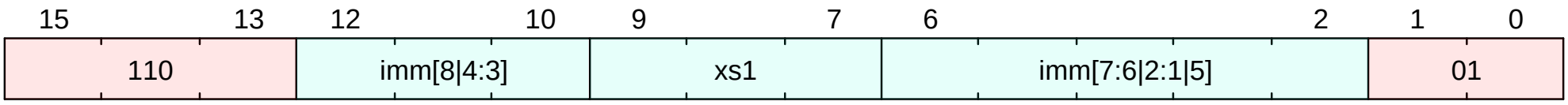
C.27. c.beqz

Branch if Equal Zero

This instruction is defined by:

Zca

C.27.1. Encoding



C.27.2. Description

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register xs1' is zero. It expands to beq <code>xs1, x0, offset</code>.

C.27.3. Access

M	S	U
Always	Always	Always

C.27.4. Decode Variables

```
signed Bits<9> imm = sext({$encoding[12], $encoding[6:5], $encoding[2], $encoding[11:10], $encoding[4:3], 1'd0});
Bits<3> xs1 = $encoding[9:7];
```

C.27.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (X[creg2reg(xs1)] == 0) {
  jump($pc + $signed(imm));
}
```

C.27.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(0);
  let taken : bool = match op {
    RISCV_BEQ  => rs1_val == rs2_val,
    RISCV_BNE  => rs1_val != rs2_val,
    RISCV_BLT  => rs1_val <_s rs2_val,
    RISCV_BGE  => rs1_val >=_s rs2_val,
    RISCV_BLTU => rs1_val <_u rs2_val,
    RISCV_BGEU => rs1_val >=_u rs2_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:

- IllegalInstruction
- InstructionAddressMisaligned

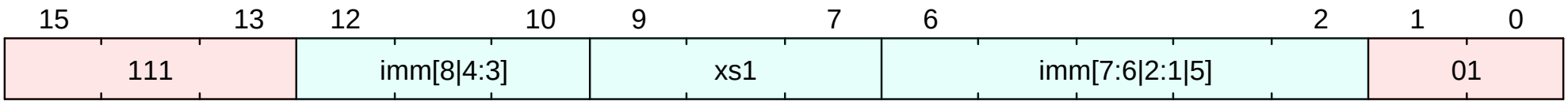
C.28. c.bnez

Branch if NOT Equal Zero

This instruction is defined by:

Zca

C.28.1. Encoding



C.28.2. Description

C.BEQZ performs conditional control transfers. The offset is sign-extended and added to the pc to form the branch target address. It can therefore target a ±256 B range. C.BEQZ takes the branch if the value in register xs1' is NOT zero. It expands to beq <code>xs1, x0, offset</code>.

C.28.3. Access

M	S	U
Always	Always	Always

C.28.4. Decode Variables

```
signed Bits<9> imm = sext({$encoding[12], $encoding[6:5], $encoding[2], $encoding[11:10], $encoding[4:3], 1'd0});
Bits<3> xs1 = $encoding[9:7];
```

C.28.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (X[creg2reg(xs1)] != 0) {
  jump($pc + $signed(imm));
}
```

C.28.6. Sail Operation

```
{
  let rs1_val = X(rs1);
  let rs2_val = X(0);
  let taken : bool = match op {
    RISCV_BEQ  => rs1_val == rs2_val,
    RISCV_BNE  => rs1_val != rs2_val,
    RISCV_BLT  => rs1_val <_s rs2_val,
    RISCV_BGE  => rs1_val >=_s rs2_val,
    RISCV_BLTU => rs1_val <_u rs2_val,
    RISCV_BGEU => rs1_val >=_u rs2_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:

- IllegalInstruction
- InstructionAddressMisaligned

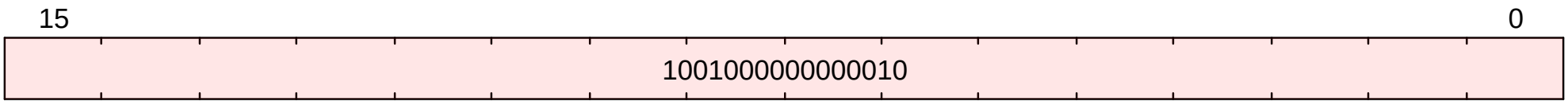
C.29. c.ebreak

Breakpoint exception

This instruction is defined by:

Zca

C.29.1. Encoding



C.29.2. Description

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

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

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

C.29.3. Access

M	S	U
Always	Always	Always

C.29.4. Decode Variables

```
```

C.29.5. IDL Operation

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

C.29.6. Sail Operation

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

C.29.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint
- IllegalInstruction

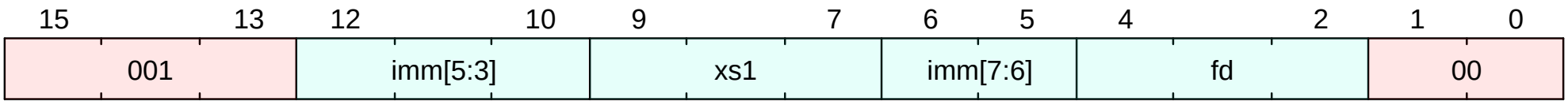
C.30. c.fld

Load double-precision

This instruction is defined by:

Zcd

C.30.1. Encoding



C.30.2. Description

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

C.30.3. Access

M	S	U
Always	Always	Always

C.30.4. Decode Variables

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

C.30.5. IDL Operation

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

C.30.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

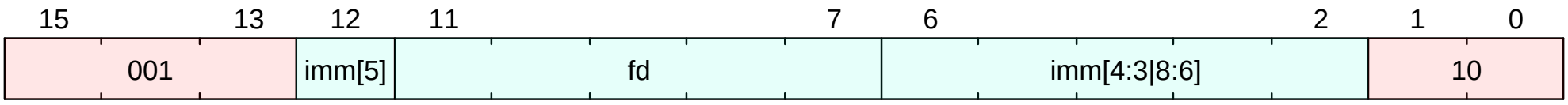
C.31. c.fldsp

Load doubleword into floating-point register from stack

This instruction is defined by:

Zcd

C.31.1. Encoding



C.31.2. Description

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

C.31.3. Access

M	S	U
Always	Always	Always

C.31.4. Decode Variables

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

C.31.5. IDL Operation

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

C.31.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

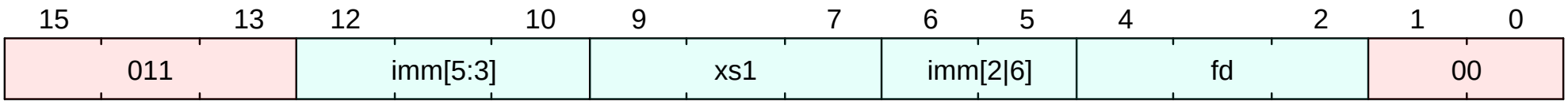
C.32. c.flw

Load single-precision

This instruction is defined by:

Zcf

C.32.1. Encoding



C.32.2. Description

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

C.32.3. Access

M	S	U
Always	Always	Always

C.32.4. Decode Variables

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

C.32.5. IDL Operation

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

C.32.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

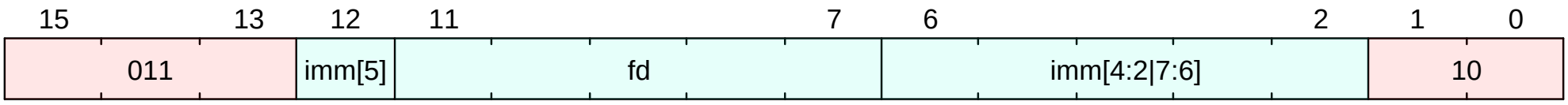
C.33. c.flwsp

Load word into floating-point register from stack

This instruction is defined by:

Zcf

C.33.1. Encoding



C.33.2. Description

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

C.33.3. Access

M	S	U
Always	Always	Always

C.33.4. Decode Variables

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

C.33.5. IDL Operation

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

C.33.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

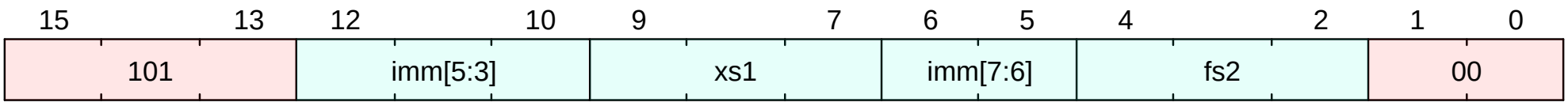
C.34. c.fsd

Store double-precision

This instruction is defined by:

Zcd

C.34.1. Encoding



C.34.2. Description

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

C.34.3. Access

M	S	U
Always	Always	Always

C.34.4. Decode Variables

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

C.34.5. IDL Operation

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

C.34.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

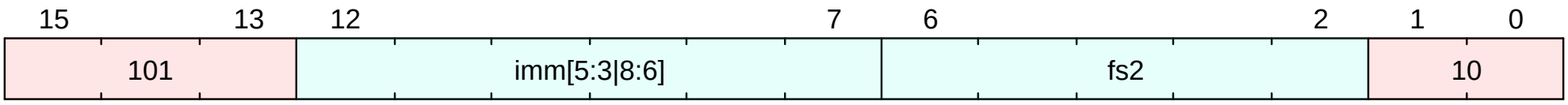
C.35. c.fsdsp

Store double-precision value to stack

This instruction is defined by:

Zcd

C.35.1. Encoding



C.35.2. Description

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

C.35.3. Access

M	S	U
Always	Always	Always

C.35.4. Decode Variables

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

C.35.5. IDL Operation

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

C.35.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

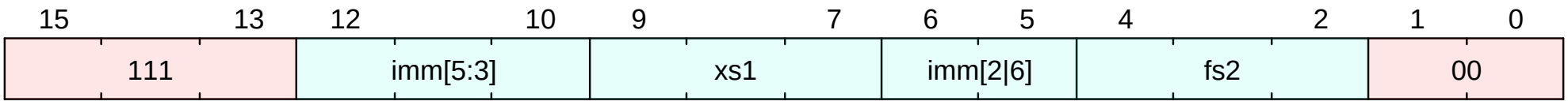
C.36. c.fsw

Store single-precision

This instruction is defined by:

Zcf

C.36.1. Encoding



C.36.2. Description

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

C.36.3. Access

M	S	U
Always	Always	Always

C.36.4. Decode Variables

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

C.36.5. IDL Operation

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

C.36.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

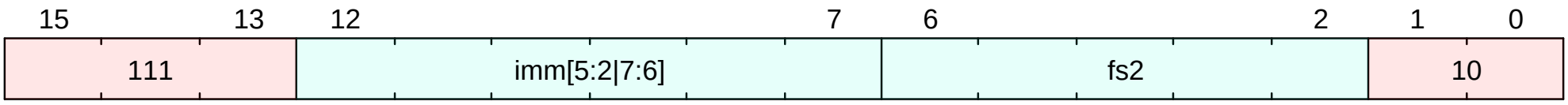
C.37. c.fswsp

Store single-precision value to stack

This instruction is defined by:

Zcf

C.37.1. Encoding



C.37.2. Description

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

C.37.3. Access

M	S	U
Always	Always	Always

C.37.4. Decode Variables

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

C.37.5. IDL Operation

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

C.37.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

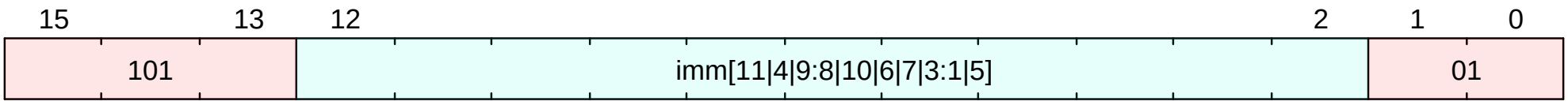
C.38. c.j

Jump

This instruction is defined by:

Zca

C.38.1. Encoding



C.38.2. Description

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

C.38.3. Access

M	S	U
Always	Always	Always

C.38.4. Decode Variables

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

C.38.5. IDL Operation

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

C.38.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

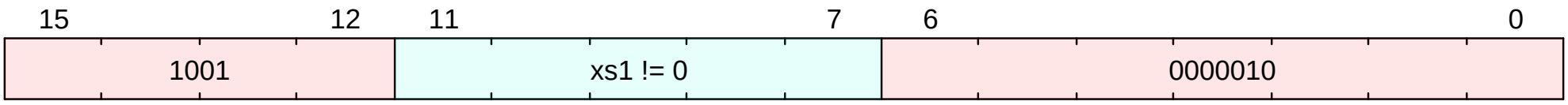
C.39. c.jalr

Jump and Link Register

This instruction is defined by:

Zca

C.39.1. Encoding



C.39.2. Description

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump (pc+2) to the link register, x1. C.JALR expands to jalr x1, 0(xs1).

C.39.3. Access

M	S	U
Always	Always	Always

C.39.4. Decode Variables

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

C.39.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg addr = X[xs1];
XReg returnaddr;
returnaddr = $pc + 2;
X[1] = returnaddr;
jump(addr);
```

C.39.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

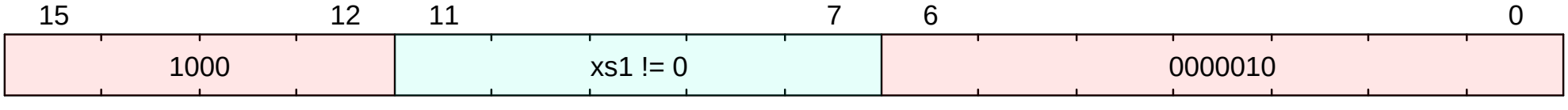
C.40. c.jr

Jump Register

This instruction is defined by:

Zca

C.40.1. Encoding



C.40.2. Description

C.JR (jump register) performs an unconditional control transfer to the address in register xs1. C.JR expands to jalr x0, 0(xs1).

C.40.3. Access

M	S	U
Always	Always	Always

C.40.4. Decode Variables

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

C.40.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
jump(X[xs1]);
```

C.40.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- InstructionAddressMisaligned

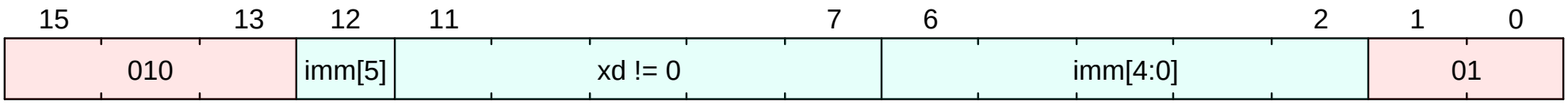
C.41. c.li

Load the sign-extended 6-bit immediate

This instruction is defined by:

Zca

C.41.1. Encoding



C.41.2. Description

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

C.41.3. Access

M	S	U
Always	Always	Always

C.41.4. Decode Variables

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

C.41.5. IDL Operation

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

C.41.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.42. c.lui

Load Upper Immediate

This instruction is defined by:

Zca

C.42.1. Encoding

C.42.2. Description

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

C.42.3. Access

M	S	U
Always	Always	Always

C.42.4. Decode Variables

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

C.42.5. IDL Operation

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

C.42.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

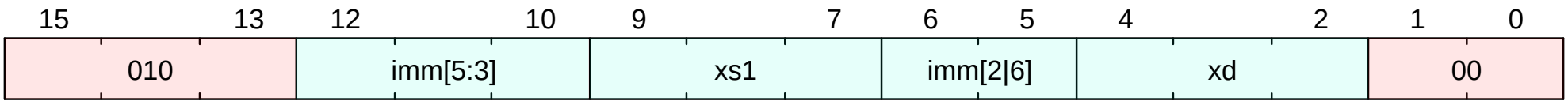
C.43. c.lw

Load word

This instruction is defined by:

Zca

C.43.1. Encoding



C.43.2. Description

Loads a 32-bit value from memory into register xd. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register xs1. It expands to lw `xd, offset(xs1)`.

C.43.3. Access

M	S	U
Always	Always	Always

C.43.4. Decode Variables

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

C.43.5. IDL Operation

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

C.43.6. 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(paddr, _) =>
          match (width) {
            BYTE =>
              process_load(rd, vaddr, mem_read(Read(Data), paddr, 1, aq, rl, false), is_unsigned),
            HALF =>
              process_load(rd, vaddr, mem_read(Read(Data), paddr, 2, aq, rl, false), is_unsigned),
            WORD =>
              process_load(rd, vaddr, mem_read(Read(Data), paddr, 4, aq, rl, false), is_unsigned),
            DOUBLE if sizeof(xlen) >= 64 =>
              process_load(rd, vaddr, mem_read(Read(Data), paddr, 8, aq, rl, false), is_unsigned),
            _ => report_invalid_width(__FILE__, __LINE__, width, "load")
          }
      }
  }
}
```

C.43.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

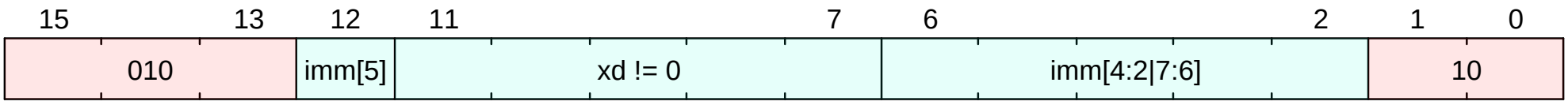
C.44. c.lwsp

Load word from stack pointer

This instruction is defined by:

Zca

C.44.1. Encoding



C.44.2. Description

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

C.44.3. Access

M	S	U
Always	Always	Always

C.44.4. Decode Variables

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

C.44.5. IDL Operation

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

C.44.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

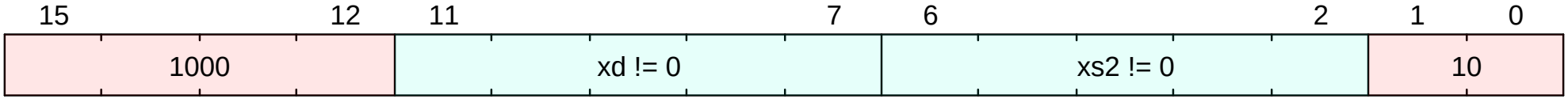
C.45. c.mv

Move Register

This instruction is defined by:

Zca

C.45.1. Encoding



C.45.2. Description

C.MV (move register) performs copy of the data in register xs2 to register xd C.MV expands to addi xd, x0, xs2.

C.45.3. Access

M	S	U
Always	Always	Always

C.45.4. Decode Variables

```
Bits<5> xd = $encoding[11:7];
Bits<5> xs2 = $encoding[6:2];
```

C.45.5. IDL Operation

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

C.45.6. Sail Operation

```
{
  let xs2_val = X(xs2);
  X(rs) = xs2_val
  RETIRE_SUCCESS
}
```

C.45.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

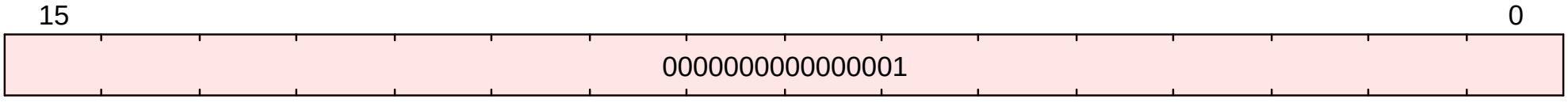
C.46. c.nop

Non-operation

This instruction is defined by:

Zca

C.46.1. Encoding



C.46.2. Description

C.NOP expands into `addi x0, x0, 0`.

C.46.3. Access

M	S	U
Always	Always	Always

C.46.4. Decode Variables

```
```

C.46.5. IDL Operation

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

C.46.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

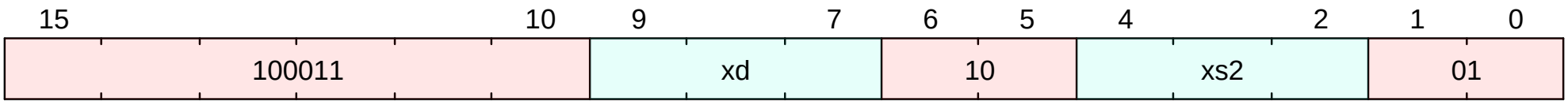
C.47. c.or

Or

This instruction is defined by:

Zca

C.47.1. Encoding



C.47.2. Description

Or xd with xs2, and store the result in xd The xd and xs2 register indexes should be used as xd+8 and xs2+8 (registers x8-x15). C.OR expands into `or xd, xd, xs2`.

C.47.3. Access

M	S	U
Always	Always	Always

C.47.4. Decode Variables

```
Bits<3> xs2 = $encoding[4:2];
Bits<3> xd = $encoding[9:7];
```

C.47.5. IDL Operation

```
XReg t0 = X[creg2reg(xd)];
XReg t1 = X[creg2reg(xs2)];
X[creg2reg(xd)] = t0 | t1;
```

C.47.6. Sail Operation

```
{
  let rs1_val = X(rd+8);
  let rs2_val = X(rs2+8);
  let result : xlenbits = match op {
    RISCV_ADD  => rs1_val + rs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(rs1_val <_s rs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(rs1_val <_u rs2_val)),
    RISCV_AND  => rs1_val & rs2_val,
    RISCV_OR   => rs1_val | rs2_val,
    RISCV_XOR  => rs1_val ^ rs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then rs1_val << (rs2_val[4..0])
                  else rs1_val << (rs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then rs1_val >> (rs2_val[4..0])
                  else rs1_val >> (rs2_val[5..0]),
    RISCV_SUB  => rs1_val - rs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(rs1_val, rs2_val[4..0])
                  else shift_right_arith64(rs1_val, rs2_val[5..0])
  };
  X(rd+8) = result;
  RETIRE_SUCCESS
}
```

C.47.7. Exceptions

This instruction does not generate synchronous exceptions.

C.48. c.slli

Shift left logical immediate

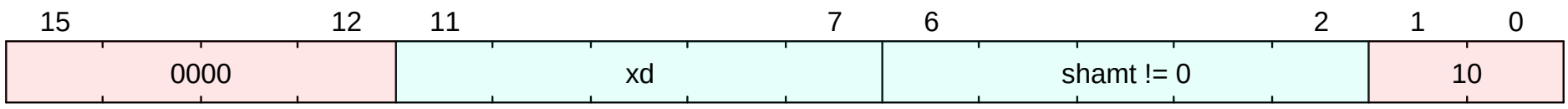
This instruction is defined by:

Zca

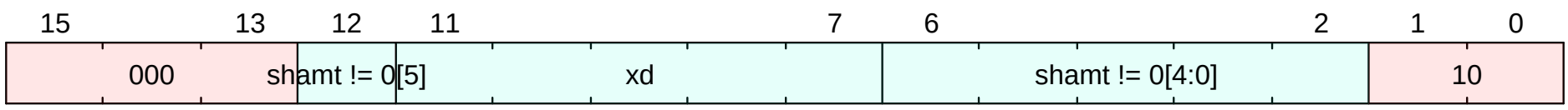
C.48.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.48.2. Description

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

C.48.3. Access

M	S	U
Always	Always	Always

C.48.4. Decode Variables

RV32

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

RV64

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

C.48.5. IDL Operation

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

C.48.6. Sail Operation

```
{
  let rd_val = X(rd);
  /* the decoder guard should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then rd_val << shamt[4..0]
                  else rd_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then rd_val >> shamt[4..0]
                  else rd_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(rd_val, shamt[4..0])
                  else shift_right_arith64(rd_val, shamt)
  };
  X(rd) = result;
  RETIRE_SUCCESS
```

```
}
```

C.48.7. Exceptions

This instruction does not generate synchronous exceptions.

C.49. c.srai

Shift right arithmetical immediate

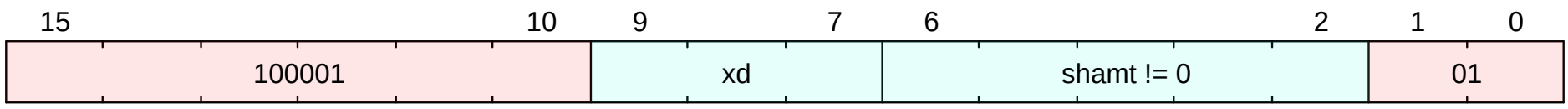
This instruction is defined by:

Zca

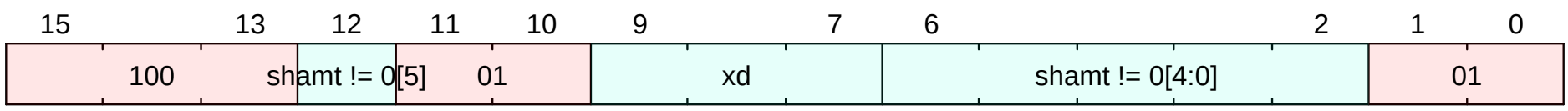
C.49.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.49.2. Description

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

C.49.3. Access

M	S	U
Always	Always	Always

C.49.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[6:2];
Bits<3> xd = $encoding[9:7];
```

RV64

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

C.49.5. IDL Operation

```
X[creg2reg(xd)] = X[creg2reg(xd)] >>> shamt;
```

C.49.6. Sail Operation

```
{
  let rd_val = X(rd+8);
  /* the decoder guard should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then rd_val << shamt[4..0]
                  else rd_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then rd_val >> shamt[4..0]
                  else rd_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(rd_val, shamt[4..0])
                  else shift_right_arith64(rd_val, shamt)
  };
  X(rd+8) = result;
```

```
  RETIRE_SUCCESS
}
```

C.49.7. Exceptions

This instruction does not generate synchronous exceptions.

C.50. c.srli

Shift right logical immediate

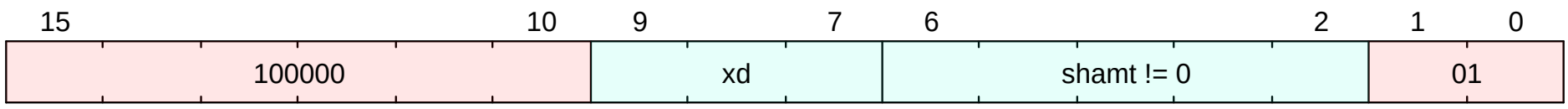
This instruction is defined by:

Zca

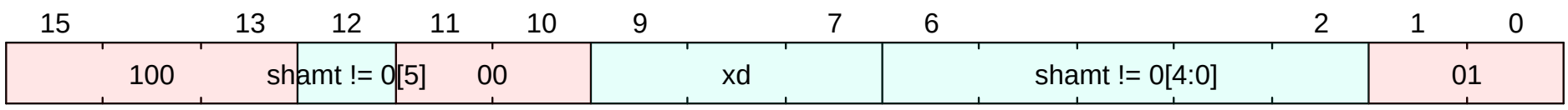
C.50.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.50.2. Description

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

C.50.3. Access

M	S	U
Always	Always	Always

C.50.4. Decode Variables

RV32

```
Bits<5> shamt = $encoding[6:2];
Bits<3> xd = $encoding[9:7];
```

RV64

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

C.50.5. IDL Operation

```
X[creg2reg(xd)] = X[creg2reg(xd)] >> shamt;
```

C.50.6. Sail Operation

```
{
  let rd_val = X(rd+8);
  /* the decoder guard should ensure that shamt[5] = 0 for RV32 */
  let result : xlenbits = match op {
    RISCV_SLLI => if   sizeof(xlen) == 32
                  then rd_val << shamt[4..0]
                  else rd_val << shamt,
    RISCV_SRLI => if   sizeof(xlen) == 32
                  then rd_val >> shamt[4..0]
                  else rd_val >> shamt,
    RISCV_SRAI => if   sizeof(xlen) == 32
                  then shift_right_arith32(rd_val, shamt[4..0])
                  else shift_right_arith64(rd_val, shamt)
  };
  X(rd+8) = result;
```

```
  RETIRE_SUCCESS
}
```

C.50.7. Exceptions

This instruction does not generate synchronous exceptions.

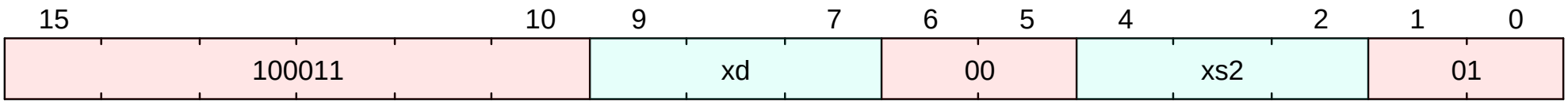
C.51. c.sub

Subtract

This instruction is defined by:

Zca

C.51.1. Encoding



C.51.2. Description

Subtract the value in xs2 from xd, and store the result in xd. The xd and xs2 register indexes should be used as xd+8 and xs2+8 (registers x8-x15). C.SUB expands into `sub xd, xd, xs2`.

C.51.3. Access

M	S	U
Always	Always	Always

C.51.4. Decode Variables

```
Bits<3> xs2 = $encoding[4:2];
Bits<3> xd = $encoding[9:7];
```

C.51.5. IDL Operation

```
XReg t0 = X[creg2reg(xd)];
XReg t1 = X[creg2reg(xs2)];
X[creg2reg(xd)] = t0 - t1;
```

C.51.6. Sail Operation

```
{
  let rs1_val = X(rd+8);
  let rs2_val = X(rs2+8);
  let result : xlenbits = match op {
    RISCV_ADD  => rs1_val + rs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(rs1_val <_s rs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(rs1_val <_u rs2_val)),
    RISCV_AND  => rs1_val & rs2_val,
    RISCV_OR   => rs1_val | rs2_val,
    RISCV_XOR  => rs1_val ^ rs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then rs1_val << (rs2_val[4..0])
                  else rs1_val << (rs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then rs1_val >> (rs2_val[4..0])
                  else rs1_val >> (rs2_val[5..0]),
    RISCV_SUB  => rs1_val - rs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(rs1_val, rs2_val[4..0])
                  else shift_right_arith64(rs1_val, rs2_val[5..0])
  };
  X(rd+8) = result;
  RETIRE_SUCCESS
}
```

C.51.7. Exceptions

This instruction does not generate synchronous exceptions.

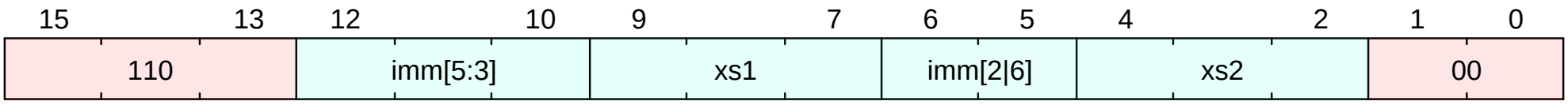
C.52. c.sw

Store word

This instruction is defined by:

Zca

C.52.1. Encoding



C.52.2. Description

Stores a 32-bit value in register xs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the base address in register xs1. It expands to sw `rs2, offset(xs1)`.

C.52.3. Access

M	S	U
Always	Always	Always

C.52.4. Decode Variables

```
Bits<7> imm = {$encoding[5], $encoding[12:10], $encoding[6], 2'd0};
Bits<3> xs2 = $encoding[4:2];
Bits<3> xs1 = $encoding[9:7];
```

C.52.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[creg2reg(xs1)] + imm;
write_memory<32>(virtual_address, X[creg2reg(xs2)][31:0], $encoding);
```

C.52.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

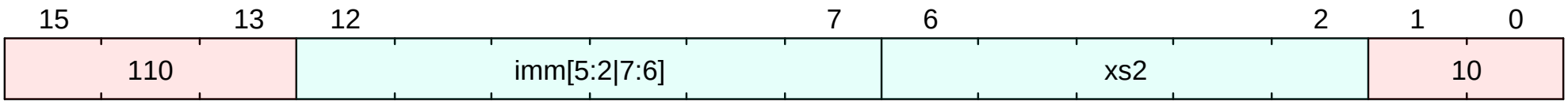
C.53. c.swsp

Store word to stack

This instruction is defined by:

Zca

C.53.1. Encoding



C.53.2. Description

Stores a 32-bit value in register xs2 to memory. It computes an effective address by adding the zero-extended offset, scaled by 4, to the stack pointer, x2. It expands to sw `xs2, offset(x2)`.

C.53.3. Access

M	S	U
Always	Always	Always

C.53.4. Decode Variables

```
Bits<8> imm = {$encoding[8:7], $encoding[12:9], 2'd0};
Bits<5> xs2 = $encoding[6:2];
```

C.53.5. IDL Operation

```
if (implemented?(ExtensionName::C) && (CSR[misa].C == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg virtual_address = X[2] + imm;
write_memory<32>(virtual_address, X[xs2][31:0], $encoding);
```

C.53.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

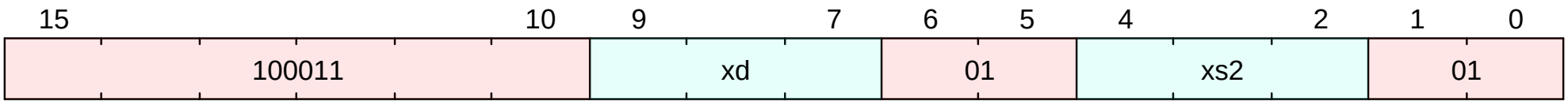
C.54. c.xor

Exclusive Or

This instruction is defined by:

Zca

C.54.1. Encoding



C.54.2. Description

Exclusive or xd with xs2, and store the result in xd The xd and xs2 register indexes should be used as xd+8 and xs2+8 (registers x8-x15). C.XOR expands into `xor xd, xd, xs2`.

C.54.3. Access

M	S	U
Always	Always	Always

C.54.4. Decode Variables

```
Bits<3> xs2 = $encoding[4:2];
Bits<3> xd = $encoding[9:7];
```

C.54.5. IDL Operation

```
XReg t0 = X[creg2reg(xd)];
XReg t1 = X[creg2reg(xs2)];
X[creg2reg(xd)] = t0 ^ t1;
```

C.54.6. Sail Operation

```
{
  let rs1_val = X(rd+8);
  let rs2_val = X(rs2+8);
  let result : xlenbits = match op {
    RISCV_ADD  => rs1_val + rs2_val,
    RISCV_SLT  => zero_extend(bool_to_bits(rs1_val <_s rs2_val)),
    RISCV_SLTU => zero_extend(bool_to_bits(rs1_val <_u rs2_val)),
    RISCV_AND  => rs1_val & rs2_val,
    RISCV_OR   => rs1_val | rs2_val,
    RISCV_XOR  => rs1_val ^ rs2_val,
    RISCV_SLL  => if   sizeof(xlen) == 32
                  then rs1_val << (rs2_val[4..0])
                  else rs1_val << (rs2_val[5..0]),
    RISCV_SRL  => if   sizeof(xlen) == 32
                  then rs1_val >> (rs2_val[4..0])
                  else rs1_val >> (rs2_val[5..0]),
    RISCV_SUB  => rs1_val - rs2_val,
    RISCV_SRA  => if   sizeof(xlen) == 32
                  then shift_right_arith32(rs1_val, rs2_val[4..0])
                  else shift_right_arith64(rs1_val, rs2_val[5..0])
  };
  X(rd+8) = result;
  RETIRE_SUCCESS
}
```

C.54.7. Exceptions

This instruction does not generate synchronous exceptions.

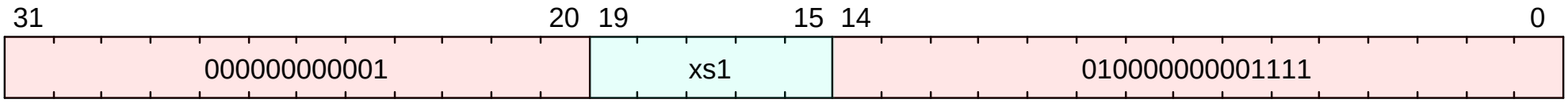
C.55. cbo.clean

Cache Block Clean

This instruction is defined by:

Zicbom

C.55.1. Encoding



C.55.2. Description

Cleans an entire cache block globally throughout the system.

Exactly what happens is coherence protocol-dependent, but in general it is expected that after this operation():

- The cache block will be in the clean (not dirty) state in any coherent cache holding a valid copy of the line.
- The data will be cleaned to a point such that an incoherent load can observe the cleaned data.

cbo.clean is ordered by FENCE instructions but not FENCE.I or SFENCE.VMA.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Both PMP and PMA access control must be the same for all bytes in the block; otherwise, cbo.clean has UNSPECIFIED behavior. <%- end -%>

Clean operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AMO Guest-Page Fault` if virtual memory translation fails during G-stage translation.
<%- end -%>
* `Store/AMO Page Fault` if virtual memory translation fails <% if ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AMO Access Fault` if a PMP or PMA access check fails
```

<%- if CACHE_BLOCK_SIZE.bit_length ⇐ [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

C.55.3. Access

M	S	U
Always	Sometimes	Sometimes

Access is controlled through menvcfg.CBZE, senvcfg.CBZE, and henvcfg.CBZE. When access is denied, the instruction either raises an `Illegal Instruction` or `Virtual Instruction` exception according to the table below.

menvcfg.CBCFE	senvcfg.CBCFE	henvcfg.CBCFE	cbo.clean Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`Illegal Instruction`	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
1	0	0	executes	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
1	1	0	executes	executes	`Virtual Instruction`	`Virtual Instruction`
1	0	1	executes	`Illegal Instruction`	executes	`Virtual Instruction`
1	1	1	executes	executes	executes	executes

C.55.4. Decode Variables

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

C.55.5. Exceptions

This instruction does not generate synchronous exceptions.

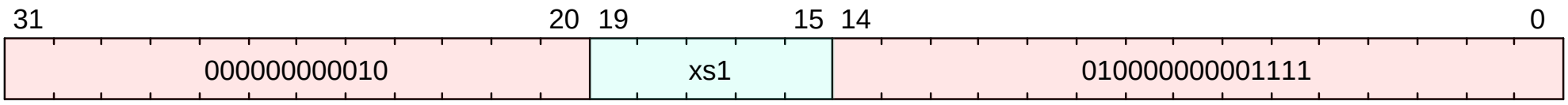
C.56. cbo.flush

Cache Block Flush

This instruction is defined by:

Zicbom

C.56.1. Encoding



C.56.2. Description

Flushes an entire cache block by cleaning it and then invalidating it in all caches.

cbo.flush is ordered by `FENCE` instructions but not `FENCE.I` or `SFENCE.VMA`.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Both PMP and PMA access control must be the same for all bytes in the block; otherwise, cbo.flush has UNSPECIFIED behavior. <%- end -%>

Flush operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AMO Guest-Page Fault` if virtual memory translation fails during G-stage translation.
<%- end -%>
* `Store/AMO Page Fault` if virtual memory translation fails <% if ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AMO Access Fault` if a PMP or PMA access check fails.
```

<%- if CACHE_BLOCK_SIZE.bit_length ⇐ [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

C.56.3. Access

M	S	U
Always	Sometimes	Sometimes

Access is controlled through menvcfg.CBZE, senvcfg.CBZE, and henvcfg.CBZE. When access is denied, the instruction either raises an `Illegal Instruction` or `Virtual Instruction` exception according to the table below.

menvcfg.CBCFE	senvcfg.CBCFE	henvcfg.CBCFE	cbo.flush Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`Illegal Instruction`	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
1	0	0	executes	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
1	1	0	executes	executes	`Virtual Instruction`	`Virtual Instruction`
1	0	1	executes	`Illegal Instruction`	executes	`Virtual Instruction`
1	1	1	executes	executes	executes	executes

C.56.4. Decode Variables

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

C.56.5. Exceptions

This instruction does not generate synchronous exceptions.

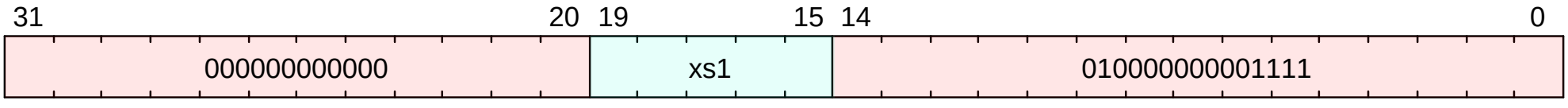
C.57. cbo.inval

Cache Block Invalidate

This instruction is defined by:

Zicbom

C.57.1. Encoding



C.57.2. Description

Either invalidates or flushes (clean + invalidate) a cache block, depending on the current mode and value of menvcfg.CBIE, senvcfg.CBIE, and/or henvcfg.CBIE.

The instruction is an invalidate (without a clean) when:

- In M-mode
- In (H)S-mode and menvcfg.CBIE == 11
- In U-mode and menvcfg.CBIE == 11 and senvcfg.CBIE == 11
- In VS-mode and menvcfg.CBIE == 11 and henvcfg.CBIE == 11
- In VU-mode and menvcfg.CBIE == 11 and henvcfg.CBIE == 11 and senvcfg.CBIE == 11

Otherwise, if the instruction does not trap (see Access section), the operation is a flush. The table below summarizes the options.

menvcfg.CBIE	senvcfg.CBIE	henvcfg.CBIE	`cbe.inval` Operation				
			M-mode	S-mode	U-mode	VS-mode	VU-mode
00	-	-	Invalidate	`Illegal Instruction`	`Illegal Instruction`	`Illegal Instruction`	`Illegal Instruction`
01	00	00	Invalidate	Flush	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
01	00	01	Invalidate	Flush	`Illegal Instruction`	Flush	`Virtual Instruction`
01	00	11	Invalidate	Flush	`Illegal Instruction`	Flush	`Virtual Instruction`
01	01	00	Invalidate	Flush	Flush	`Virtual Instruction`	`Virtual Instruction`
01	01	01	Invalidate	Flush	Flush	Flush	Flush
01	01	11	Invalidate	Flush	Flush	Flush	Flush
01	11	00	Invalidate	Flush	Flush	`Virtual Instruction`	`Virtual Instruction`
01	11	01	Invalidate	Flush	Flush	Flush	Flush
01	11	11	Invalidate	Flush	Flush	Flush	Flush
11	00	00	Invalidate	Invalidate	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
11	00	01	Invalidate	Invalidate	`Illegal Instruction`	Flush	`Virtual Instruction`
11	00	11	Invalidate	Invalidate	`Illegal Instruction`	Invalidate	`Virtual Instruction`
11	01	00	Invalidate	Invalidate	Flush	`Virtual Instruction`	`Virtual Instruction`
11	01	01	Invalidate	Invalidate	Flush	Flush	Flush
11	01	11	Invalidate	Invalidate	Flush	Invalidate	Flush
11	11	00	Invalidate	Invalidate	Invalidate	`Virtual Instruction`	`Virtual Instruction`
11	11	01	Invalidate	Invalidate	Invalidate	Flush	Flush
11	11	11	Invalidate	Invalidate	Invalidate	Invalidate	Invalidate

cbo.inval is ordered by `FENCE` instructions but not `FENCE.I` or `SFENCE.VMA`.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Both PMP and PMA access control must be the same for all bytes in the block; otherwise, cbo.zero has UNSPECIFIED behavior. <%- end -%>

Invalidate operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AMO Guest-Page Fault` if virtual memory translation fails during G-stage translation.
<%- end -%>
* `Store/AMO Page Fault` if virtual memory translation fails <% if ext?(:H) %>when V=0 or during VS-stage translation<% end %>
```

```
* `Store/AMO Access Fault` if a PMP or PMA access check fails.
```

<%- if CACHE_BLOCK_SIZE.bit_length ⇐ [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

C.57.3. Access

M	S	U
Always	Sometimes	Sometimes

Access is controlled through menvcfg.CBIE, senvcfg.CBIE, and henvcfg.CBIE. When access is denied, the instruction either raises an `Illegal Instruction` or `Virtual Instruction` exception according to the table below.

`xenvcfg.CBIE` == 10 is reserved, and cannot be written by software. As such, that pattern is excluded from the table below.

menvcfg.CBIE	senvcfg.CBIE	henvcfg.CBIE	cbo.inval Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
00	-	-	`Illegal Instruction`	`Illegal Instruction`	`Illegal Instruction`	`Illegal Instruction`
01/11	00	00	executes	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
01/11	01/11	00	executes	executes	`Virtual Instruction`	`Virtual Instruction`
01/11	00	01/11	executes	`Illegal Instruction`	executes	`Virtual Instruction`
01/11	01/11	01/11	executes	executes	executes	executes

C.57.4. Decode Variables

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

C.57.5. Exceptions

This instruction does not generate synchronous exceptions.

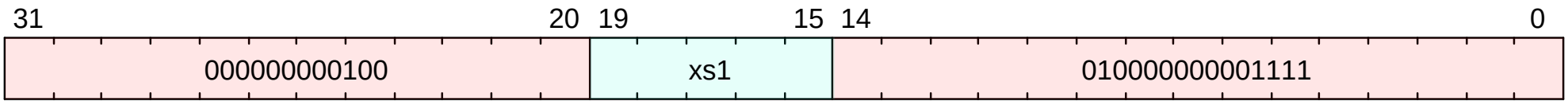
C.58. cbo.zero

Cache Block Zero

This instruction is defined by:

Zicboz

C.58.1. Encoding



C.58.2. Description

Zeros an entire cache block

The block zeroing does not need to be atomic.

cbo.zero is ordered by `FENCE` instructions but not `FENCE.I` or `SFENCE.VMA`.

<%- if CACHE_BLOCK_SIZE.bit_length > [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Both PMP and PMA access control must be the same for all bytes in the block; otherwise, cbo.zero has UNSPECIFIED behavior. <%- end -%>

Clean operations are treated as stores for page and access permissions. If permission checks fail, one of the following exceptions will occur:

```
<%- if ext?(:H) -%>
* `Store/AMO Guest-Page Fault` if virtual memory translation fails during G-stage translation.
<%- end -%>
* `Store/AMO Page Fault` if virtual memory translation fails <% if ext?(:H) %>when V=0 or during VS-stage translation<% end %>
* `Store/AMO Access Fault` if a PMP or PMA access check fails.
```

<%- if CACHE_BLOCK_SIZE.bit_length ⇐ [PMP_GRANULARITY, PMA_GRANULARITY].min -%> Because cache blocks are naturally aligned and always fit in a single PMP or PMA regions, the PMP and PMA access checks only need to check a single address in the line. <%- end -%>

CBO operations never raise a misaligned address fault.

C.58.3. Access

M	S	U
Always	Sometimes	Sometimes

Access is controlled through menvcfg.CBZE, senvcfg.CBZE, and henvcfg.CBZE. When access is denied, the instruction either raises an `Illegal Instruction` or `Virtual Instruction` exception according to the table below.

menvcfg.CBZE	senvcfg.CBZE	henvcfg.CBZE	cbo.zero Instruction Behavior			
			S-mode	U-mode	VS-mode	VU-mode
0	-	-	`Illegal Instruction`	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
1	0	0	executes	`Illegal Instruction`	`Virtual Instruction`	`Virtual Instruction`
1	1	0	executes	executes	`Virtual Instruction`	`Virtual Instruction`
1	0	1	executes	`Illegal Instruction`	executes	`Virtual Instruction`
1	1	1	executes	executes	executes	executes

C.58.4. Decode Variables

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

C.58.5. IDL Operation

```
if ((mode() == PrivilegeMode::M && CSR[menvcfg].CBZE == 0) || (mode() == PrivilegeMode::U && CSR[senvcfg].CBZE == 0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
} else if ((mode() == PrivilegeMode::VS && CSR[henvcfg].CBZE == 0) || (mode() == PrivilegeMode::VU && (CSR[henvcfg].CBZE |
CSR[senvcfg].CBZE) == 0)) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
} else {
```

```
  XReg mask = CACHE_BLOCK_SIZE - 1;
  XReg cache_block_vaddr = X[xs1] & ~mask;
  TranslationResult result;
  result = translate(cache_block_vaddr, MemoryOperation::Write, effective_ldst_mode(), $encoding);
  access_check(result.paddr, CACHE_BLOCK_SIZE * 8, cache_block_vaddr, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
  cache_block_zero(result.paddr);
}
```

C.58.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoPageFault
- VirtualInstruction

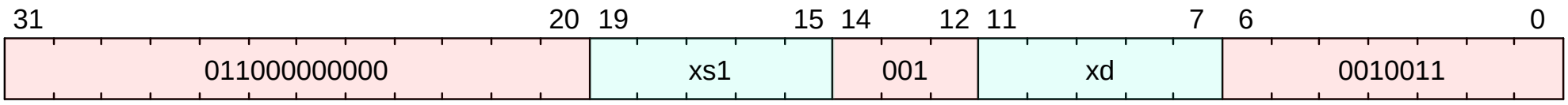
C.59. clz

Count leading zero bits

This instruction is defined by:

Zbb

C.59.1. Encoding



C.59.2. Description

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

C.59.3. Access

M	S	U
Always	Always	Always

C.59.4. Decode Variables

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

C.59.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[xd] = (xlen() - 1) - $signed(highest_set_bit(X[xs1]));
```

C.59.6. Sail Operation

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

C.59.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

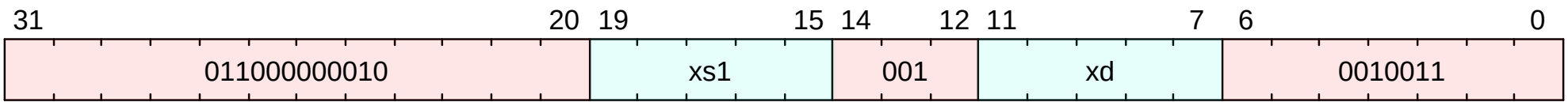
C.60. cpop

Count set bits

This instruction is defined by:

Zbb

C.60.1. Encoding



C.60.2. Description

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

Listing 1. Software Hint

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

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

C.60.3. Access

M	S	U
Always	Always	Always

C.60.4. Decode Variables

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

C.60.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg bitcount = 0;
XReg xs1_val = X[xs1];
for (U32 i = 0; i < xlen(); i++) {
  if (xs1_val[i] == 1'b1) {
    bitcount = bitcount + 1;
  }
}
X[xd] = bitcount;
```

C.60.6. Sail Operation

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

C.60.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

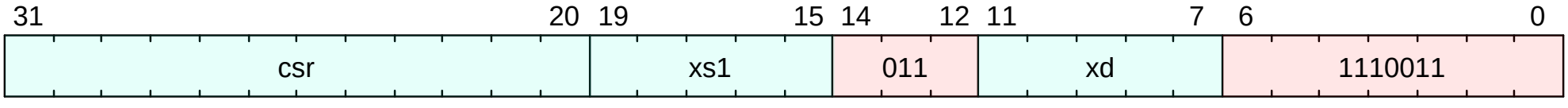
C.61. csrrc

Atomic Read and Clear Bits in CSR

This instruction is defined by:

Zicsr

C.61.1. Encoding



C.61.2. Description

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

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

C.61.3. Access

M	S	U
Always	Always	Always

C.61.4. Decode Variables

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

C.61.5. IDL Operation

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

C.61.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

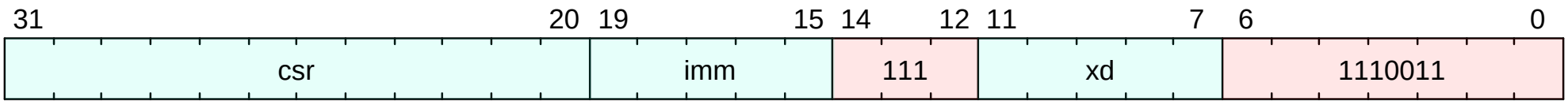
C.62. csrrci

Atomic Read and Clear Bits in CSR with Immediate

This instruction is defined by:

Zicsr

C.62.1. Encoding



C.62.2. Description

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

C.62.3. Access

M	S	U
Always	Always	Always

C.62.4. Decode Variables

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

C.62.5. IDL Operation

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

C.62.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

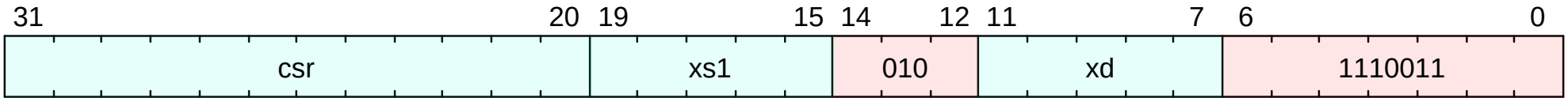
C.63. csrrs

Atomic Read and Set Bits in CSR

This instruction is defined by:

Zicsr

C.63.1. Encoding



C.63.2. Description

Atomically read and set bits in a CSR.

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

C.63.3. Access

M	S	U
Always	Always	Always

C.63.4. Decode Variables

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

C.63.5. IDL Operation

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

C.63.6. Sail Operation

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

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

C.63.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

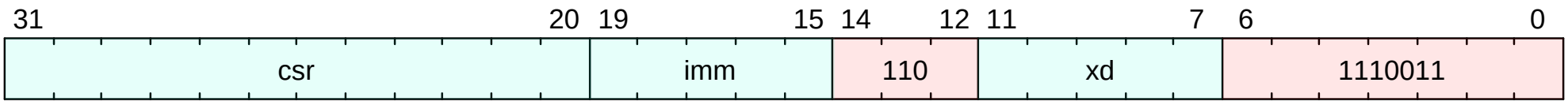
C.64. csrrsi

Atomic Read and Set Bits in CSR with Immediate

This instruction is defined by:

Zicsr

C.64.1. Encoding



C.64.2. Description

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

C.64.3. Access

M	S	U
Always	Always	Always

C.64.4. Decode Variables

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

C.64.5. IDL Operation

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

C.64.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

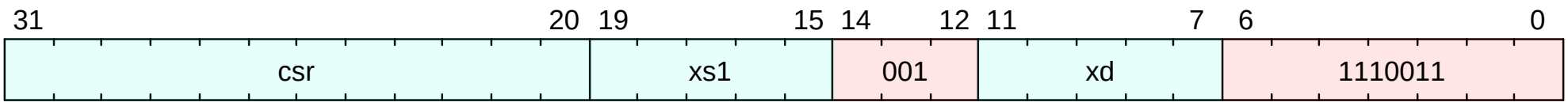
C.65. csrrw

Atomic Read/Write CSR

This instruction is defined by:

Zicsr

C.65.1. Encoding



C.65.2. Description

Atomically swap values in the CSRs and integer registers.

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

C.65.3. Access

M	S	U
Always	Always	Always

C.65.4. Decode Variables

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

C.65.5. IDL Operation

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

C.65.6. Sail Operation

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

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

C.65.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

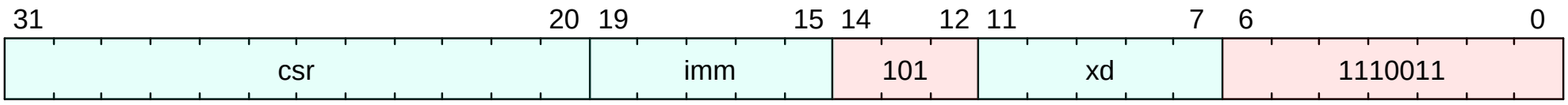
C.66. csrrwi

Atomic Read/Write CSR Immediate

This instruction is defined by:

Zicsr

C.66.1. Encoding



C.66.2. Description

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

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

C.66.3. Access

M	S	U
Always	Always	Always

C.66.4. Decode Variables

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

C.66.5. IDL Operation

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

C.66.6. Sail Operation

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

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

C.66.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

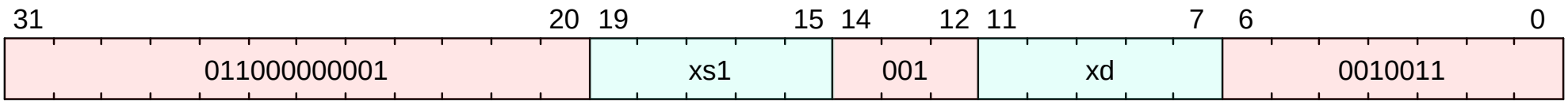
C.67. ctz

Count trailing zero bits

This instruction is defined by:

Zbb

C.67.1. Encoding



C.67.2. Description

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

C.67.3. Access

M	S	U
Always	Always	Always

C.67.4. Decode Variables

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

C.67.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xlen() == 32) {
  X[xd] = lowest_set_bit(X[xs1][31:0]);
} else {
  X[xd] = lowest_set_bit(X[xs1]);
}
```

C.67.6. Sail Operation

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

C.67.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

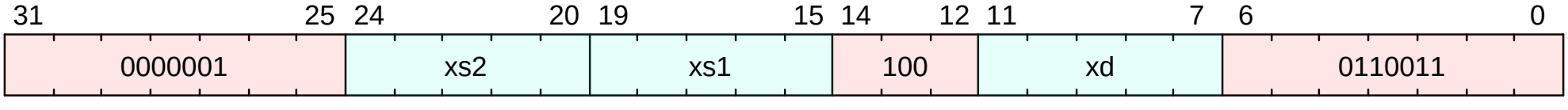
C.68. div

Signed division

This instruction is defined by:

M

C.68.1. Encoding



C.68.2. Description

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

Division by zero will put -1 into xd.

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

C.68.3. Access

M	S	U
Always	Always	Always

C.68.4. Decode Variables

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

C.68.5. IDL Operation

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

C.68.6. 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.68.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

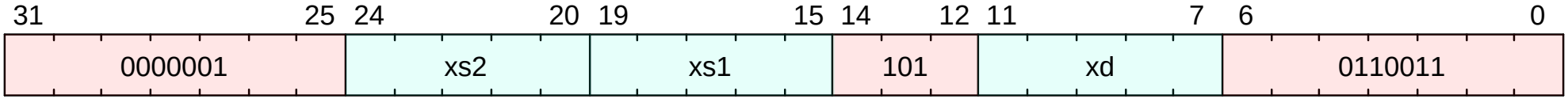
C.69. divu

Unsigned division

This instruction is defined by:

M

C.69.1. Encoding



C.69.2. Description

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

The remainder is discarded.

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

C.69.3. Access

M	S	U
Always	Always	Always

C.69.4. Decode Variables

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

C.69.5. IDL Operation

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

C.69.6. 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.69.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

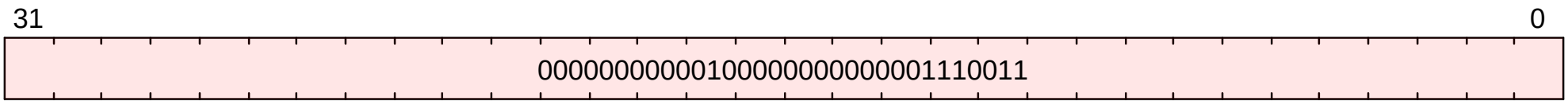
C.70. ebreak

Breakpoint exception

This instruction is defined by:

I

C.70.1. Encoding



C.70.2. 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.70.3. Access

M	S	U
Always	Always	Always

C.70.4. Decode Variables

```
```

C.70.5. IDL Operation

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

C.70.6. Sail Operation

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

C.70.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Breakpoint

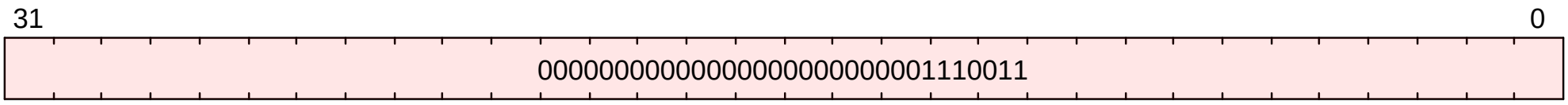
C.71. ecall

Environment call

This instruction is defined by:

I

C.71.1. Encoding



C.71.2. Description

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

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

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

C.71.3. Access

M	S	U
Always	Always	Always

C.71.4. Decode Variables

```
```

C.71.5. 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.71.6. 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.71.7. Exceptions

This instruction may result in the following synchronous exceptions:

- Mcall
- Scall
- Ucall
- VScall

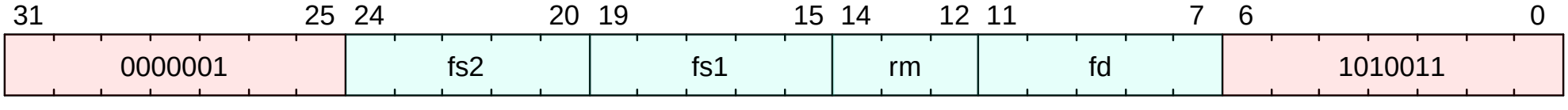
C.72. fadd.d

Floating-Point Add Double-Precision

This instruction is defined by:

(D Zdinx)

C.72.1. Encoding



C.72.2. Description

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

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. Exceptions

This instruction does not generate synchronous exceptions.

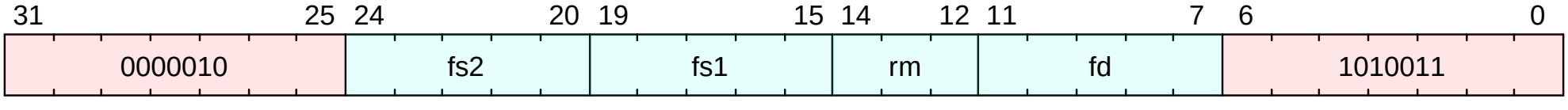
C.73. fadd.h

No synopsis available

This instruction is defined by:

(Zfh || Zhinx)

C.73.1. Encoding



C.73.2. Description

No description available.

C.73.3. Access

M	S	U
Always	Always	Always

C.73.4. 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.73.5. IDL Operation

```

```

C.73.6. Exceptions

This instruction does not generate synchronous exceptions.

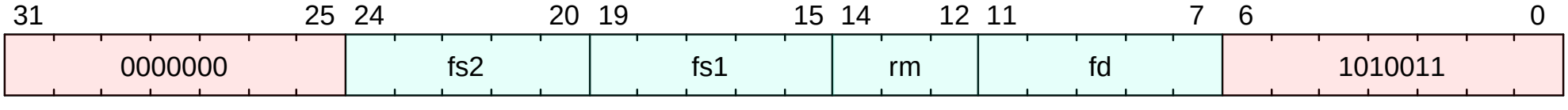
C.74. fadd.s

Floating-Point Add Single-Precision

This instruction is defined by:

F

C.74.1. Encoding



C.74.2. Description

The fadd.s instruction performs single-precision floating-point addition of `fs1` and `fs2` and writes the final result to `fd`.

C.74.3. Access

M	S	U
Always	Always	Always

C.74.4. 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.74.5. IDL Operation

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

C.74.6. 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.74.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

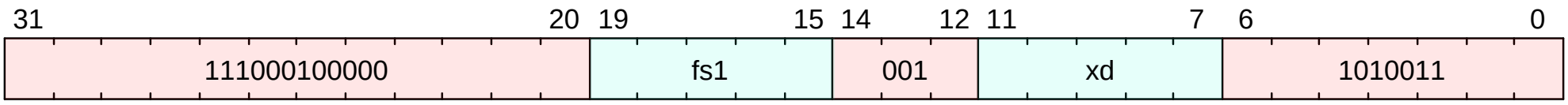
C.75. fclass.d

Floating-Point Classify Double-Precision

This instruction is defined by:

(D ⍰ Zdinx)

C.75.1. Encoding



C.75.2. Description

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

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

Table 10. Format of result of fclass instruction.

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

C.75.3. Access

M	S	U
Always	Always	Always

C.75.4. Decode Variables

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

C.75.5. IDL Operation

```

```

C.75.6. Exceptions

This instruction does not generate synchronous exceptions.

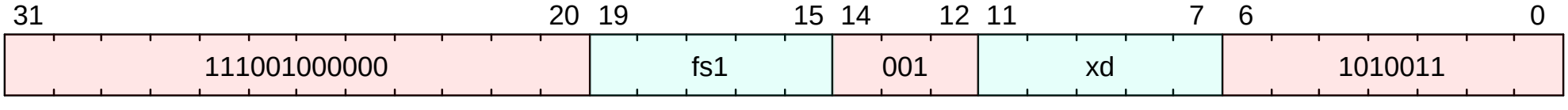
C.76. fclass.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

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

C.76.5. IDL Operation

```

```

C.76.6. Exceptions

This instruction does not generate synchronous exceptions.

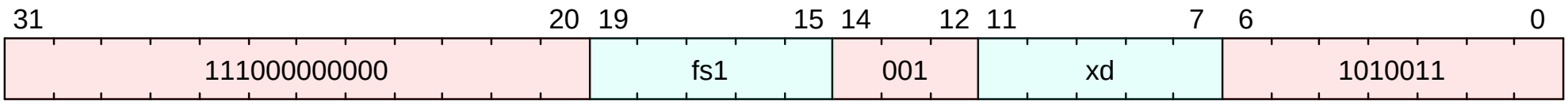
C.77. fclass.s

Floating-Point Classify Single-Precision

This instruction is defined by:

F

C.77.1. Encoding



C.77.2. Description

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

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

Table 11. Format of result of `fclass` instruction.

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

C.77.3. Access

M	S	U
Always	Always	Always

C.77.4. Decode Variables

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

C.77.5. IDL Operation

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

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

C.77.6. 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.77.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

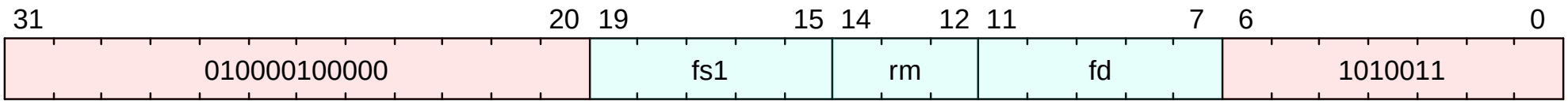
C.78. fcvt.d.s

Floating-Point Convert Single-Precision to Double-Precision

This instruction is defined by:

(D Zdinx)

C.78.1. Encoding



C.78.2. Description

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

C.78.3. Access

M	S	U
Always	Always	Always

C.78.4. Decode Variables

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

C.78.5. IDL Operation

```

```

C.78.6. Exceptions

This instruction does not generate synchronous exceptions.

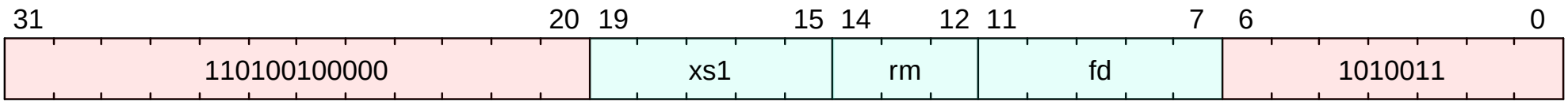
C.79. fcvt.d.w

Floating-Point Convert Word to Double-Precision

This instruction is defined by:

(D ∪ Zdinx)

C.79.1. Encoding



C.79.2. Description

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

C.79.3. Access

M	S	U
Always	Always	Always

C.79.4. Decode Variables

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

C.79.5. IDL Operation

```

```

C.79.6. Exceptions

This instruction does not generate synchronous exceptions.

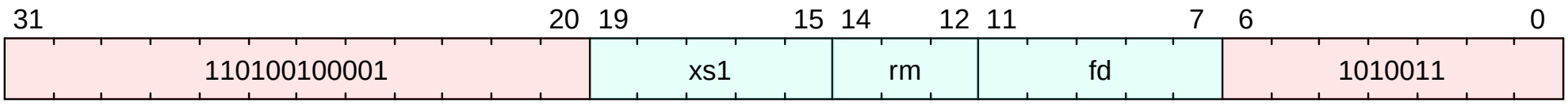
C.80. fcvt.d.wu

Floating-Point Convert Unsigned Word to Double-Precision

This instruction is defined by:

(D Zdinx)

C.80.1. Encoding



C.80.2. Description

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

C.80.3. Access

M	S	U
Always	Always	Always

C.80.4. Decode Variables

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

C.80.5. IDL Operation

```

```

C.80.6. Exceptions

This instruction does not generate synchronous exceptions.

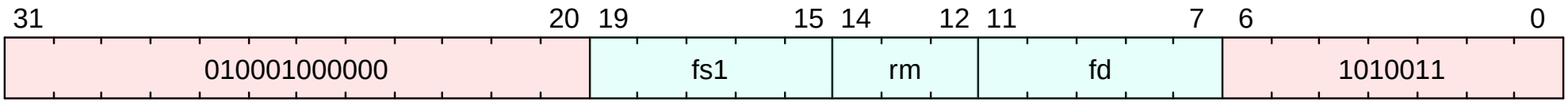
C.81. fcvt.h.s

Convert half-precision float to a single-precision float

This instruction is defined by:

(Zfhmin ▯ Zhinxmin)

C.81.1. Encoding



C.81.2. Description

Converts a half-precision number in floating-point register *fs1* into a single-precision floating-point number in floating-point register *fd*.

fcvt.h.s rounds according to the *rm* field.

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.81.3. Access

M	S	U
Always	Always	Always

C.81.4. Decode Variables

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

C.81.5. IDL Operation

```
check_f_ok($encoding);
Bits<16> hp_value = f[fs1][15:0];
Bits<1> sign = hp_value[15];
Bits<5> exp = hp_value[14:10];
Bits<10> frac = hp_value[9:0];
if (exp == 0x1F) {
  if (frac != 0) {
    if ((hp_value & 0x0200) != 0) {
      set_fp_flag(FpFlag::NV);
    }
    f[fd] = HP_CANONICAL_NAN;
  } else {
    f[fd] = packToF32UI(sign, 0xFF, 0);
  }
} else {
  if (exp != 0) {
    if (frac != 0) {
      f[fd] = packToF32UI(sign, 0, 0);
    } else {
      Bits<6> norm_exp;
      (norm_exp, frac = softfloat_normSubnormalF16Sig(frac));
      exp = norm_exp - 1;
      f[fd] = packToF32UI(sign, exp + 0x70, frac << 13);
    }
  } else {
    f[fd] = packToF32UI(sign, exp + 0x70, frac << 13);
  }
}
mark_f_state_dirty();
```

C.81.6. 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_H) = riscv_ui64ToF16 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_H(rd) = rd_val_H;
      RETIRE_SUCCESS
    }
  }
}
```

C.81.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

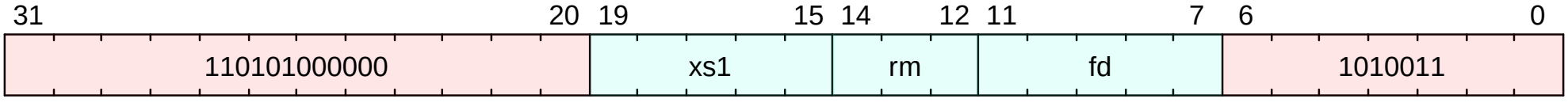
C.82. fcvt.h.w

Floating-point Convert Word to Half-precision

This instruction is defined by:

(Zfh Zhinx)

C.82.1. Encoding



C.82.2. Description

fcvt.h.w converts a 32-bit signed integer to a half-precision floating-point number.

C.82.3. Access

M	S	U
Always	Always	Always

C.82.4. Decode Variables

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

C.82.5. IDL Operation

```

```

C.82.6. Exceptions

This instruction does not generate synchronous exceptions.

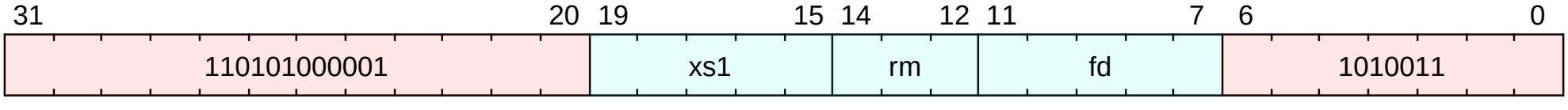
C.83. fcvt.h.wu

Floating-point Convert Unsigned Word to Half-precision

This instruction is defined by:

(Zfh Zhinx)

C.83.1. Encoding



C.83.2. Description

fcvt.h.wu converts a 32-bit unsigned integer to a half-precision floating-point number.

C.83.3. Access

M	S	U
Always	Always	Always

C.83.4. Decode Variables

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

C.83.5. IDL Operation

```

```

C.83.6. Exceptions

This instruction does not generate synchronous exceptions.

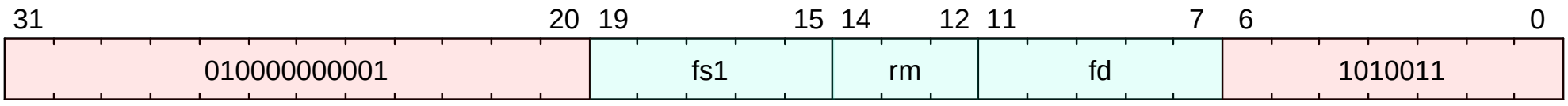
C.84. fcvt.s.d

Floating-Point Convert Double-Precision to Single-Precision

This instruction is defined by:

(D ∥ Zdinx)

C.84.1. Encoding



C.84.2. Description

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

C.84.3. Access

M	S	U
Always	Always	Always

C.84.4. Decode Variables

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

C.84.5. IDL Operation

```

```

C.84.6. Exceptions

This instruction does not generate synchronous exceptions.

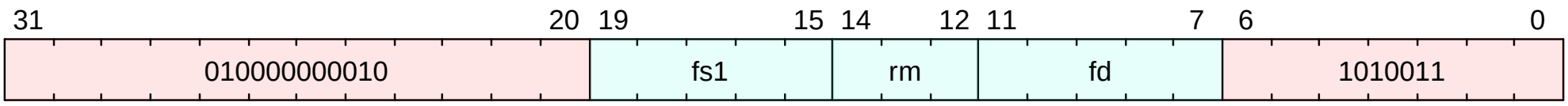
C.85. fcvt.s.h

Convert single-precision float to a half-precision float

This instruction is defined by:

(Zfhmin Zhinxmin)

C.85.1. Encoding



C.85.2. Description

Converts a single-precision number in floating-point register *fs1* into a half-precision floating-point number in floating-point register *fd*.

fcvt.s.h will never round, and so the 'rm' field is effectively ignored.

C.85.3. Access

M	S	U
Always	Always	Always

C.85.4. Decode Variables

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

C.85.5. 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> frac = sp_value[22:0];
if (exp == 0xFF) {
  if (frac != 0) {
    if ((sp_value & 0x00400000) != 0) {
      set_fp_flag(FpFlag::NV);
    }
    f[fd] = nan_box<16, FLEN>(HP_CANONICAL_NAN);
  } else {
    f[fd] = nan_box<16, FLEN>(packToF16UI(sign, 0x1F, 0));
  }
} else {
  Bits<16> frac16 = (frac >> 9) | frac & 0x1ff) != 0 ? 1 : 0);   if ((exp | frac16) == 0) {     f[fd] = nan_box<16,
FLEN>(packToF16UI(sign, 0, 0;
  } else {
    assert(false, "TODO: implement roundPackToF16");
  }
}
mark_f_state_dirty();
```

C.85.6. 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_H) = riscv_ui64ToF16 (rm_3b, rs1_val_LU);

      accrue_fflags(fflags);
      F_or_X_H(rd) = rd_val_H;
```

```
      RETIRE_SUCCESS
    }
  }
}
```

C.85.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

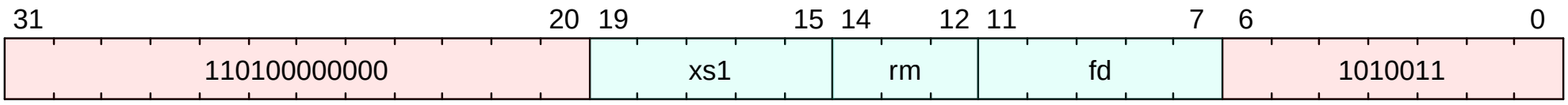
C.86. fcvt.s.w

Floating-Point Convert Word to Single-Precision

This instruction is defined by:

F

C.86.1. Encoding



C.86.2. Description

The fcvt.s.w instruction converts a 32-bit signed integer in integer register `xs1` into a floating-point number in floating-point register `fd`.

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w fd, x0`, which will never set any exception flags.

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

C.86.3. Access

M	S	U
Always	Always	Always

C.86.4. Decode Variables

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

C.86.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
f[fd] = i32_to_f32(X[xs1], rounding_mode);
mark_f_state_dirty();
```

C.86.6. 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.86.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

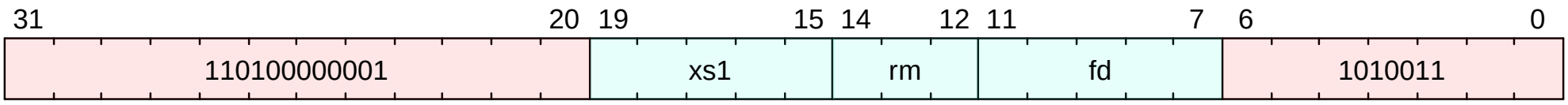
C.87. fcvt.s.wu

Floating-Point Convert Unsigned Word to Single-Precision

This instruction is defined by:

F

C.87.1. Encoding



C.87.2. Description

The fcvt.s.wu instruction converts a 32-bit unsigned integer in integer register `xs1` into a floating-point number in floating-point register `fd`.

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w rd, x0`, which will never set any exception flags.

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

C.87.3. Access

M	S	U
Always	Always	Always

C.87.4. Decode Variables

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

C.87.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
f[fd] = ui32_to_f32(X[xs1], rounding_mode);
mark_f_state_dirty();
```

C.87.6. 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.87.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

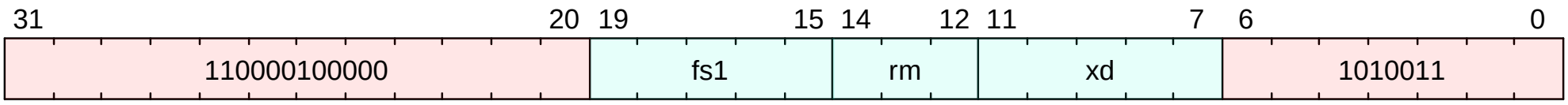
C.88. fcvt.w.d

Floating-Point Convert Double-Precision to Word

This instruction is defined by:

(D ⬜ Zdinx)

C.88.1. Encoding



C.88.2. Description

The fcvt.w.d instruction converts a double-precision floating-point number in floating-point register fs1 to a signed 32-bit integer, in integer register xd.

C.88.3. Access

M	S	U
Always	Always	Always

C.88.4. Decode Variables

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

C.88.5. IDL Operation

```

```

C.88.6. Exceptions

This instruction does not generate synchronous exceptions.

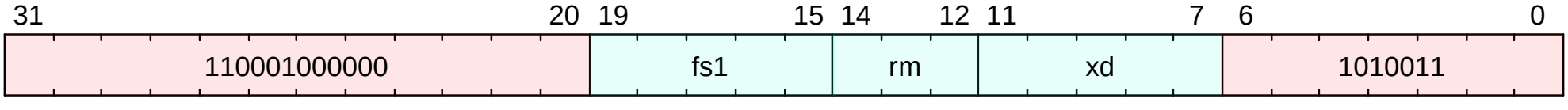
C.89. fcvt.w.h

Floating-point Convert Half-precision to Word

This instruction is defined by:

(Zfh Zhinx)

C.89.1. Encoding



C.89.2. Description

fcvt.w.h converts a half-precision floating-point number to a signed 32-bit integer.

C.89.3. Access

M	S	U
Always	Always	Always

C.89.4. Decode Variables

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

C.89.5. IDL Operation

```

```

C.89.6. Exceptions

This instruction does not generate synchronous exceptions.

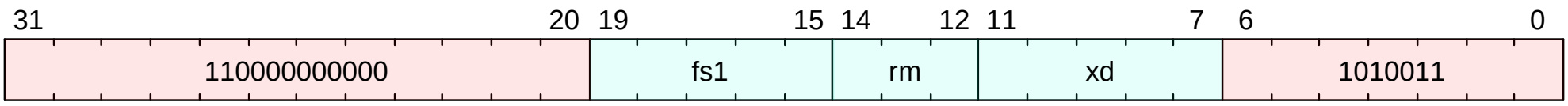
C.90. fcvt.w.s

Floating-Point Convert Single-Precision to Word

This instruction is defined by:

F

C.90.1. Encoding



C.90.2. Description

The fcvt.w.s instruction converts a floating-point number in floating-point register `fs1` to a signed 32-bit integer in integer register `xd`. For XLEN >32, fcvt.w.s sign-extends the 32-bit result to the destination register width.

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

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

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

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w xd, x0`, which will never set any exception flags.

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

C.90.3. Access

M	S	U
Always	Always	Always

C.90.4. Decode Variables

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

C.90.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
X[xd] = f32_to_i32(f[fs1], rounding_mode);
```

C.90.6. 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.90.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

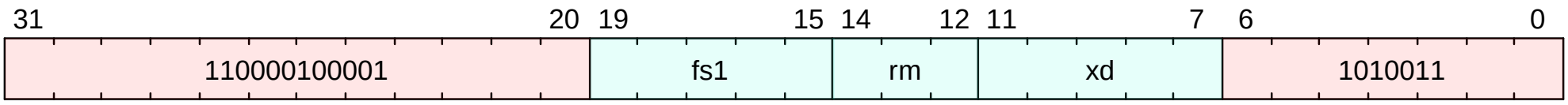
C.91. fcvt.wu.d

Floating-Point Convert Double-Precision to Unsigned Word

This instruction is defined by:

(D Zdinx)

C.91.1. Encoding



C.91.2. Description

The fcvt.wu.d instruction converts a double-precision floating-point number in floating-point register fs1 to an unsigned 32-bit integer, in integer register xd.

C.91.3. Access

M	S	U
Always	Always	Always

C.91.4. Decode Variables

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

C.91.5. IDL Operation

```

```

C.91.6. Exceptions

This instruction does not generate synchronous exceptions.

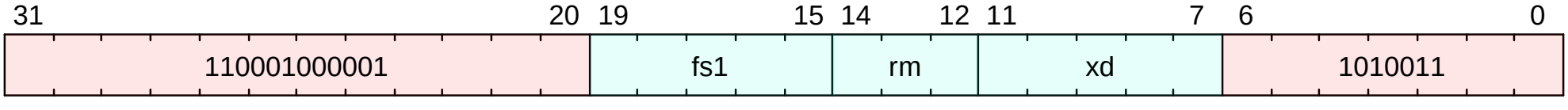
C.92. fcvt.wu.h

Floating-point Convert Half-precision to Word

This instruction is defined by:

(Zfh ▯ Zhinx)

C.92.1. Encoding



C.92.2. Description

fcvt.wu.h converts a half-precision floating-point number to a signed 32-bit unsigned integer.

C.92.3. Access

M	S	U
Always	Always	Always

C.92.4. Decode Variables

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

C.92.5. IDL Operation

```

```

C.92.6. Exceptions

This instruction does not generate synchronous exceptions.

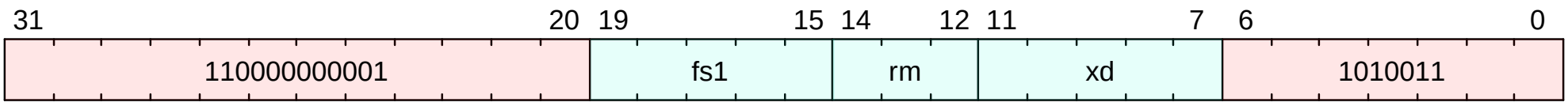
C.93. fcvt.wu.s

Floating-Point Convert Single-Precision to Unsigned Word

This instruction is defined by:

F

C.93.1. Encoding



C.93.2. Description

Converts a floating-point number in floating-point register `fs1` to an unsigned 32-bit integer in integer register `xd`. For XLEN >32, fcvt.wu.s sign-extends the 32-bit result to the destination register width.

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

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

	Value
Minimum valid input (after rounding)	`0`
Maximum valid input (after rounding)	`2^32 - 1`
Output for out-of-range negative input	`0`
Output for <code>-∞</code>	`0`
Output for out-of-range positive input	`2^32 - 1`
Output for <code>+∞</code> for <code>NaN</code>	`2^32 - 1`

All floating-point to integer and integer to floating-point conversion instructions round according to the `rm` field.

A floating-point register can be initialized to floating-point positive zero using `fcvt.s.w xd, x0`, which will never set any exception flags.

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

C.93.3. Access

M	S	U
Always	Always	Always

C.93.4. Decode Variables

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

C.93.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode rounding_mode = rm_to_mode(rm, $encoding);
X[xd] = f32_to_ui32(f[fs1], rounding_mode);
```

C.93.6. 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.93.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

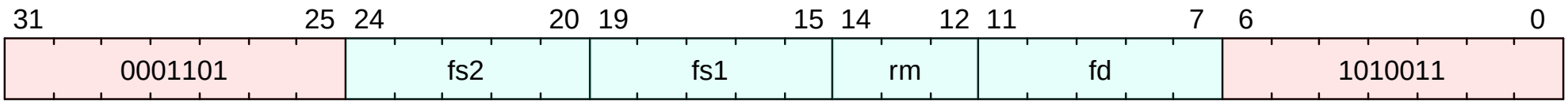
C.94. fdiv.d

Floating-Point Divide Double-Precision

This instruction is defined by:

(D ∥ Zdinx)

C.94.1. Encoding



C.94.2. Description

The fdiv.d instruction performs the double-precision floating-point division of fs1 by fs2. It is defined analogously to its single-precision counterpart, but operates on double-precision operands and produces double-precision results.

C.94.3. Access

M	S	U
Always	Always	Always

C.94.4. 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.94.5. IDL Operation

```

```

C.94.6. Exceptions

This instruction does not generate synchronous exceptions.

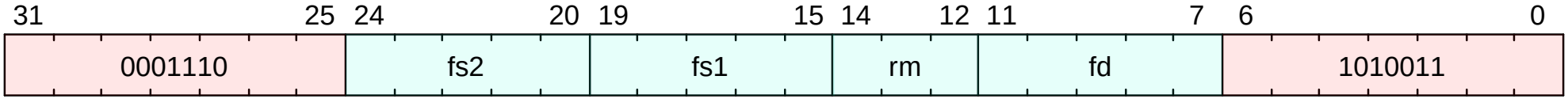
C.95. fdiv.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

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

C.95.5. IDL Operation

```

```

C.95.6. Exceptions

This instruction does not generate synchronous exceptions.

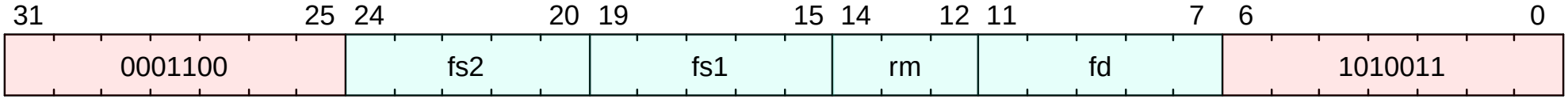
C.96. fdiv.s

Floating-Point Divide Single-Precision

This instruction is defined by:

F

C.96.1. Encoding



C.96.2. Description

The fdiv.s instruction performs the single-precision floating-point division of `fs1` by `fs2`, and writes the final result to `fd`.

C.96.3. Access

M	S	U
Always	Always	Always

C.96.4. 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.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);
  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.96.7. Exceptions

This instruction does not generate synchronous exceptions.

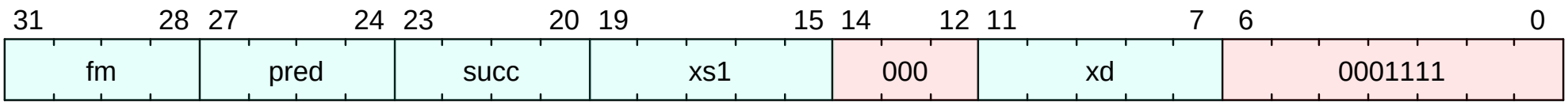
C.97. fence

Memory ordering fence

This instruction is defined by:

I

C.97.1. Encoding



C.97.2. Description

Orders memory operations.

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

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

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

Table 12. Fence mode encoding

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

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

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

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

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

Table 13. Effective PR/PW/SR/SW in (H)S-mode

menvcfg.FIOM	`pred.PI` → **effective** `PR`
	`pred.PO` → **effective** `PW`
	`succ.SI` → **effective** `SR`
	`succ.SO` → **effective** `SW`		

0	-		from encoding
1	0		from encoding
1	1		1

Table 14. Effective PR/PW/SR/SW in U-mode

menvcfg.FIOM	**senvcfg.FIOM**	`pred.PI` → **effective** `PR`
		`pred.PO` → **effective** `PW`
		`succ.SI` → **effective** `SR`
		`succ.SO` → **effective** `SW`		

0	0	-		from encoding
0	1	0		from encoding
0	1	1		1
1	-	0		from encoding

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

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

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

<%- end -%>

C.97.3. Access

M	S	U
Always	Always	Always

C.97.4. 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.97.5. IDL Operation

```
Boolean pred_i = pred[3] == 1;
Boolean pred_o = pred[2] == 1;
Boolean pred_r = pred[1] == 1;
Boolean pred_w = pred[0] == 1;
Boolean succ_i = succ[3] == 1;
Boolean succ_o = succ[2] == 1;
Boolean succ_r = succ[1] == 1;
Boolean succ_w = succ[0] == 1;
if (mode() == PrivilegeMode::S) {
  if (CSR[menvcfg].FIOM == 1) {
    if (pred_i) {
      pred_r = true;
    }
    if (pred_o) {
      pred_w = true;
    }
    if (succ_i) {
      succ_r = true;
    }
    if (succ_o) {
      succ_w = true;
    }
  }
} else if (mode() == PrivilegeMode::U) {
  if ((CSR[menvcfg].FIOM | CSR[senvcfg].FIOM) == 1) {
    if (pred_i) {
      pred_r = true;
    }
    if (pred_o) {
      pred_w = true;
    }
    if (succ_i) {
```

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

C.97.6. 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.97.7. Exceptions

This instruction does not generate synchronous exceptions.

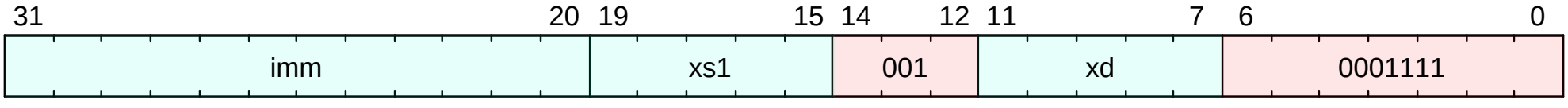
C.98. fence.i

Instruction fence

This instruction is defined by:

Zifencei

C.98.1. Encoding



C.98.2. Description

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

The unused fields in the FENCE.I instruction, *imm[11:0]*, *xs1*, and *xd*, are reserved for finer-grain fences in future extensions. For forward compatibility, base implementations shall ignore these fields, and standard software shall zero these fields.

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

C.98.3. Access

M	S	U
Always	Always	Always

C.98.4. Decode Variables

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

C.98.5. IDL Operation

```
ifence();
```

C.98.6. Sail Operation

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

C.98.7. Exceptions

This instruction does not generate synchronous exceptions.

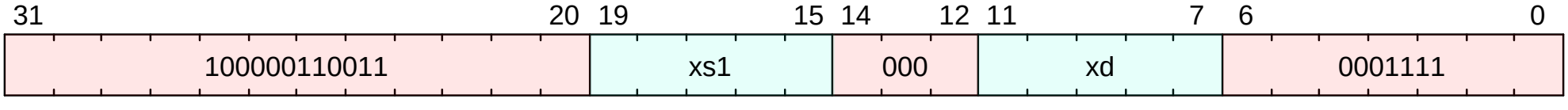
C.99. fence.tso

Memory ordering fence, total store ordering

This instruction is defined by:

I

C.99.1. Encoding



C.99.2. Description

Orders memory operations.

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

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

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

C.99.3. Access

M	S	U
Always	Always	Always

C.99.4. Decode Variables

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

C.99.5. IDL Operation

```
fence_tso();
```

C.99.6. 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.99.7. Exceptions

This instruction does not generate synchronous exceptions.

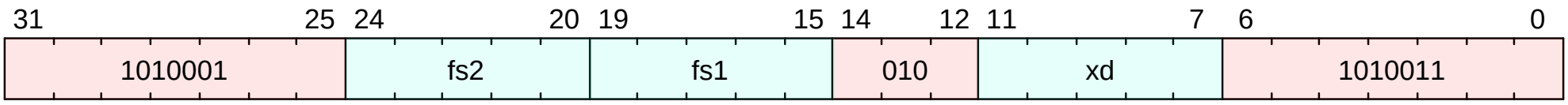
C.100. feq.d

Floating-Point Equal Double-Precision

This instruction is defined by:

(D ⊔ Zdinx)

C.100.1. Encoding



C.100.2. Description

The feq.d instruction writes 1 to xd if fs1 and fs2 are equal, and 0 otherwise. It is defined analogously to its single-precision counterpart, but operates on double-precision operands.

C.100.3. Access

M	S	U
Always	Always	Always

C.100.4. Decode Variables

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

C.100.5. IDL Operation

```

```

C.100.6. Exceptions

This instruction does not generate synchronous exceptions.

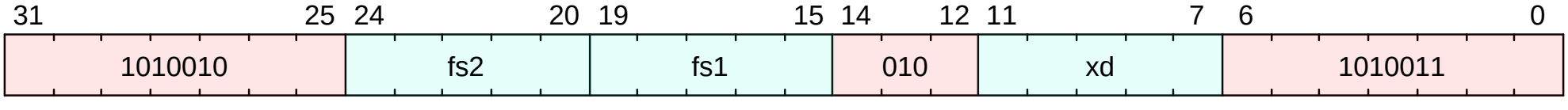
C.101. feq.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.101.1. Encoding



C.101.2. Description

No description available.

C.101.3. Access

M	S	U
Always	Always	Always

C.101.4. Decode Variables

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

C.101.5. IDL Operation

```

```

C.101.6. Exceptions

This instruction does not generate synchronous exceptions.

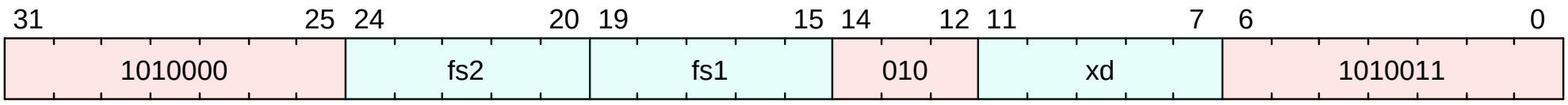
C.102. feq.s

Floating-Point Equal Single-Precision

This instruction is defined by:

F

C.102.1. Encoding



C.102.2. Description

The feq.s instruction writes 1 to `xd` if `fs1` and `fs2` are equal, and 0 otherwise. If either operand is NaN, the result is 0 (not equal). If either operand is a signaling NaN, the invalid flag is set. Positive zero is considered equal to negative zero.

C.102.3. Access

M	S	U
Always	Always	Always

C.102.4. Decode Variables

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

C.102.5. IDL Operation

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

C.102.6. 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.102.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

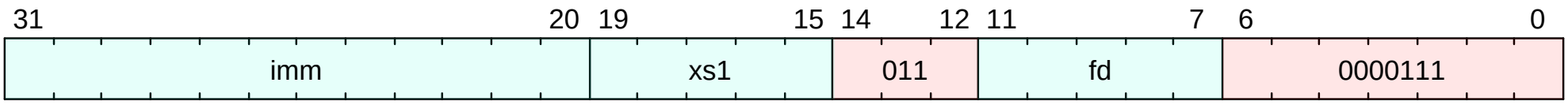
C.103. fld

Floating-Point Load Double-Precision

This instruction is defined by:

D

C.103.1. Encoding



C.103.2. Description

The fld instruction loads a double-precision floating-point value from memory into floating-point register `fd`. It is guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64. It doesn't modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

C.103.3. Access

M	S	U
Always	Always	Always

C.103.4. Decode Variables

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

C.103.5. IDL Operation

```

```

C.103.6. Exceptions

This instruction does not generate synchronous exceptions.

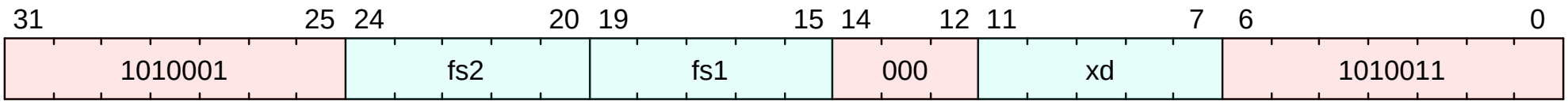
C.104. fle.d

Floating-Point Less Than or Equal Double-Precision

This instruction is defined by:

(D ⬜ Zdinx)

C.104.1. Encoding



C.104.2. Description

The fle.d instruction writes 1 to xd if fs1 is less than or equal to fs2, and 0 otherwise. It is defined analogously to its single-precision counterpart, but operates on double-precision operands.

C.104.3. Access

M	S	U
Always	Always	Always

C.104.4. Decode Variables

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

C.104.5. IDL Operation

```

```

C.104.6. Exceptions

This instruction does not generate synchronous exceptions.

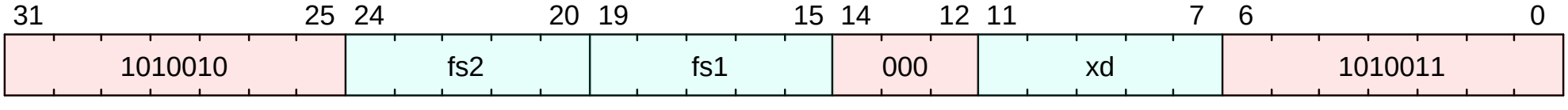
C.105. fle.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

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

C.105.5. IDL Operation

```

```

C.105.6. Exceptions

This instruction does not generate synchronous exceptions.

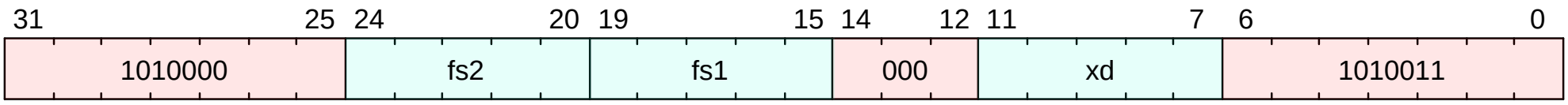
C.106. fle.s

Floating-Point Less Than or Equal Single-Precision

This instruction is defined by:

F

C.106.1. Encoding



C.106.2. Description

The fle.s instruction writes 1 to `xd` if `fs1` is less than or equal to `fs2`, and 0 otherwise. If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set. Positive zero and negative zero are considered equal.

C.106.3. Access

M	S	U
Always	Always	Always

C.106.4. Decode Variables

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

C.106.5. IDL Operation

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

C.106.6. 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.106.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

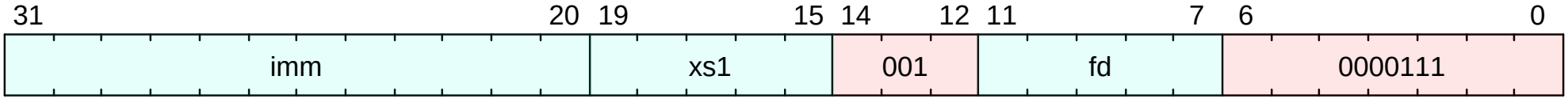
C.107. flh

Half-precision floating-point load

This instruction is defined by:

Zfhmin

C.107.1. Encoding



C.107.2. Description

The flh instruction loads a single-precision floating-point value from memory at address *xs1* + *imm* into floating-point register *xd*.

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

flh is only guaranteed to execute atomically if the effective address is naturally aligned.

C.107.3. Access

M	S	U
Always	Always	Always

C.107.4. Decode Variables

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

C.107.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[xs1] + $signed(imm);
Bits<16> hp_value = read_memory<16>(virtual_address, $encoding);
f[fd] = nan_box<16, FLEN>(hp_value);
mark_f_state_dirty();
```

C.107.6. 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.107.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

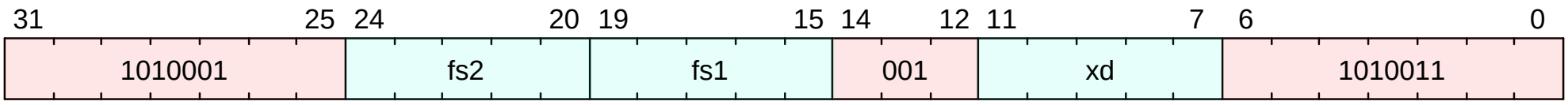
C.108. flt.d

Floating-Point Less Than Double-Precision

This instruction is defined by:

(D Zdinx)

C.108.1. Encoding



C.108.2. Description

The flt.d instruction writes 1 to xd if fs1 is less than fs2, and 0 otherwise. It is defined analogously to its single-precision counterpart, but operates on double-precision operands.

C.108.3. Access

M	S	U
Always	Always	Always

C.108.4. Decode Variables

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

C.108.5. IDL Operation

```

```

C.108.6. Exceptions

This instruction does not generate synchronous exceptions.

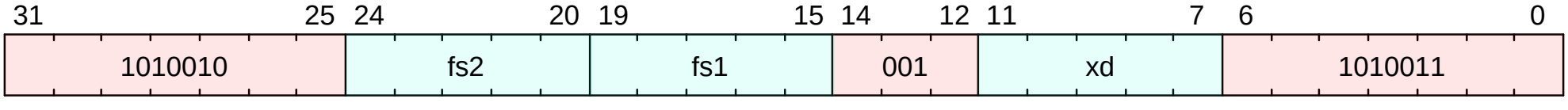
C.109. flt.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

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

C.109.5. IDL Operation

```

```

C.109.6. Exceptions

This instruction does not generate synchronous exceptions.

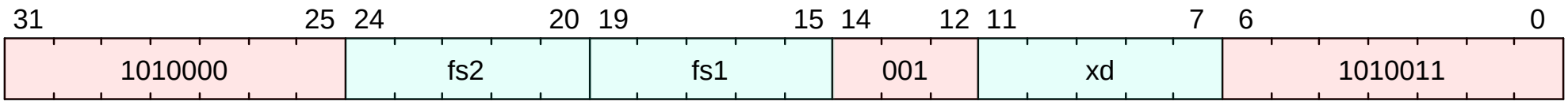
C.110. flt.s

Floating-Point Less Than Single-Precision

This instruction is defined by:

F

C.110.1. Encoding



C.110.2. Description

The flt.s instruction writes 1 to `xd` if `fs1` is less than `fs2`, and 0 otherwise. If either operand is NaN, the result is 0 (not equal). If either operand is a NaN (signaling or quiet), the invalid flag is set.

C.110.3. Access

M	S	U
Always	Always	Always

C.110.4. Decode Variables

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

C.110.5. IDL Operation

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

C.110.6. 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.110.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

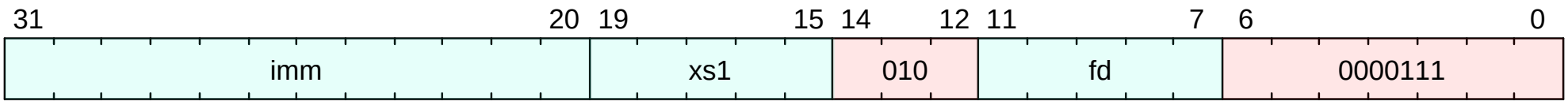
C.111. flw

Floating-Point Load Single-Precision

This instruction is defined by:

F

C.111.1. Encoding



C.111.2. Description

The flw instruction loads a single-precision floating-point value from memory at address `xs1` + `imm` into floating-point register `fd`. It does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

C.111.3. Access

M	S	U
Always	Always	Always

C.111.4. Decode Variables

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

C.111.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[xs1] + $signed(imm);
Bits<32> sp_value = read_memory<32>(virtual_address, $encoding);
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

C.111.6. 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.111.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

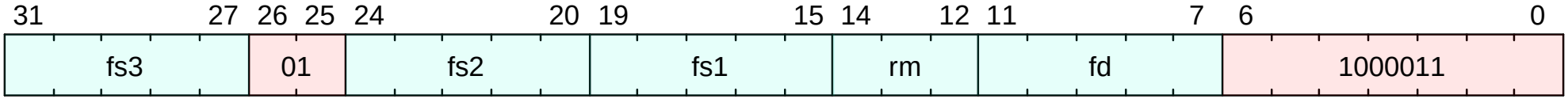
C.112. fmadd.d

Floating-Point Multiply-Add Double-Precision

This instruction is defined by:

(D Zdinx)

C.112.1. Encoding



C.112.2. Description

The fmadd.d instruction multiplies the values in fs1 and fs2, adds the value in fs3, and writes the final result to fd.

C.112.3. Access

M	S	U
Always	Always	Always

C.112.4. 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.112.5. IDL Operation

```

```

C.112.6. Exceptions

This instruction does not generate synchronous exceptions.

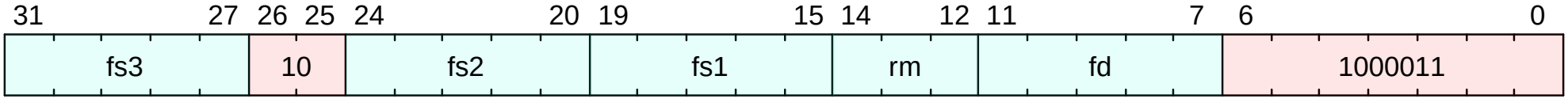
C.113. fmadd.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

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> 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.113.5. IDL Operation

```

```

C.113.6. Exceptions

This instruction does not generate synchronous exceptions.

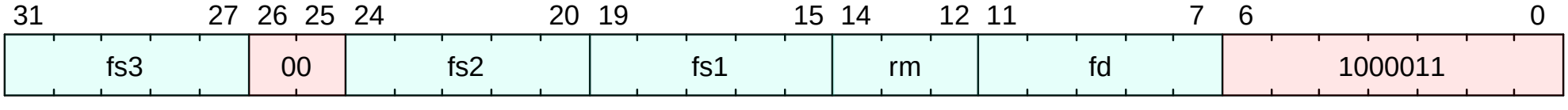
C.114. fmadd.s

Floating-Point Multiply-Add Single-Precision

This instruction is defined by:

F

C.114.1. Encoding



C.114.2. Description

The fmadd.s multiplies the values in `fs1` and `fs2`, adds the value in `fs3`, and writes the final result to `fd`.

C.114.3. Access

M	S	U
Always	Always	Always

C.114.4. 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.114.5. IDL Operation

```

```

C.114.6. 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.114.7. Exceptions

This instruction does not generate synchronous exceptions.

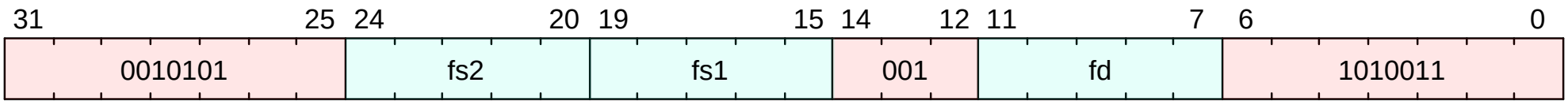
C.115. fmax.d

Floating-Point Maximum-Number Double-Precision

This instruction is defined by:

(D Zdinx)

C.115.1. Encoding



C.115.2. Description

The fmax.d instruction writes larger of fs1 and fs2 to fd. For the purposes of this instruction, the value -0.0 is considered to be less than the value +0.0. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

C.115.3. Access

M	S	U
Always	Always	Always

C.115.4. Decode Variables

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

C.115.5. IDL Operation

```

```

C.115.6. Exceptions

This instruction does not generate synchronous exceptions.

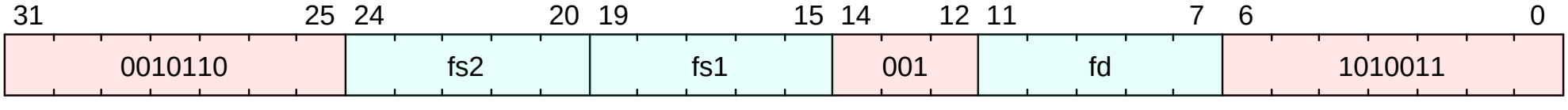
C.116. fmax.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.116.1. Encoding



C.116.2. Description

No description available.

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

```

```

C.116.6. Exceptions

This instruction does not generate synchronous exceptions.

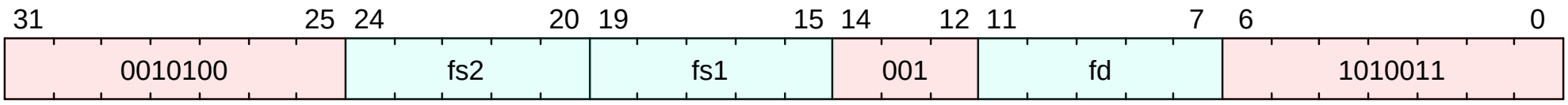
C.117. fmax.s

Floating-Point Maximum-Number Single-Precision

This instruction is defined by:

F

C.117.1. Encoding



C.117.2. Description

The fmax.s instruction writes larger of `fs1` and `fs2` to `fd`. For the purposes of this instruction, the value `-0.0` is considered to be less than the value `+0.0`. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

Note that in version 2.2 of the F extension, the fmin.s and fmax.s instructions were amended to implement the proposed *IEEE 754-201x* `minimumNumber` and `maximumNumber` operations, rather than the *IEEE 754-2008 minNum* and *maxNum* operations. These operations differ in their handling of signaling NaNs.

C.117.3. Access

M	S	U
Always	Always	Always

C.117.4. Decode Variables

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

C.117.5. IDL Operation

```

```

C.117.6. 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.117.7. Exceptions

This instruction does not generate synchronous exceptions.

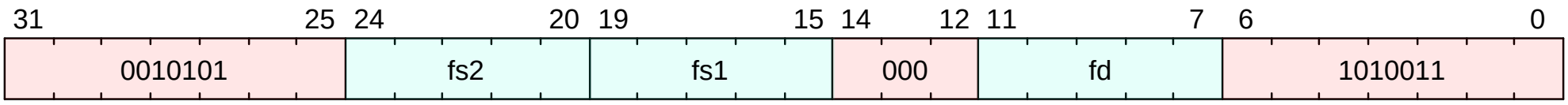
C.118. fmin.d

Floating-Point Minimum-Number Double-Precision

This instruction is defined by:

(D ⍰ Zdinx)

C.118.1. Encoding



C.118.2. Description

The fmin.d instruction writes smaller of fs1 and fs2 to fd. For the purposes of this instruction, the value -0.0 is considered to be less than the value +0.0. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

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

```

```

C.118.6. Exceptions

This instruction does not generate synchronous exceptions.

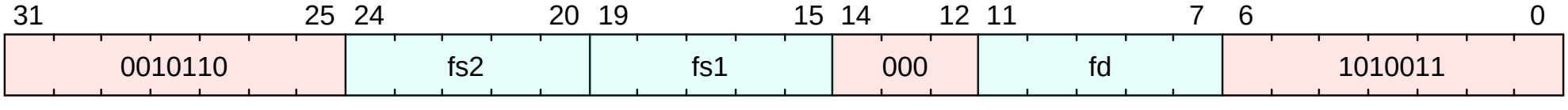
C.119. fmin.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

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

C.119.5. IDL Operation

```

```

C.119.6. Exceptions

This instruction does not generate synchronous exceptions.

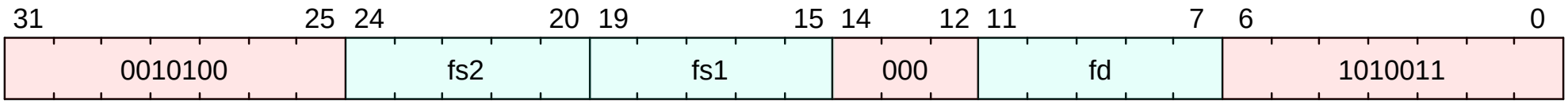
C.120. fmin.s

Floating-Point Minimum-Number Single-Precision

This instruction is defined by:

F

C.120.1. Encoding



C.120.2. Description

The fmin.s instruction writes smaller of `fs1` and `fs2` to `fd`. For the purposes of this instruction, the value `-0.0` is considered to be less than the value `+0.0`. If both inputs are NaNs, the result is the canonical NaN. If only one operand is a NaN, the result is the non-NaN operand. Signaling NaN inputs set the invalid operation exception flag, even when the result is not NaN.

Note that in version 2.2 of the F extension, the fmin.s and fmax.s instructions were amended to implement the proposed *IEEE 754-201x* `minimumNumber` and `maximumNumber` operations, rather than the *IEEE 754-2008 minNum* and *maxNum* operations. These operations differ in their handling of signaling NaNs.

C.120.3. Access

M	S	U
Always	Always	Always

C.120.4. Decode Variables

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

C.120.5. IDL Operation

```

```

C.120.6. 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.120.7. Exceptions

This instruction does not generate synchronous exceptions.

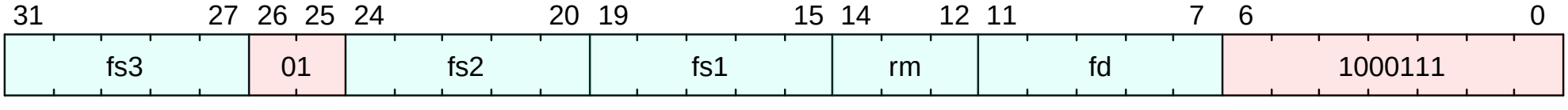
C.121. fmsub.d

Floating-Point Multiply-Subtract Double-Precision

This instruction is defined by:

(D ⬜ Zdinx)

C.121.1. Encoding



C.121.2. Description

The fmsub.d instruction multiplies the values in fs1 and fs2, subtracts the value in fs3, and writes the final result to fd.

C.121.3. Access

M	S	U
Always	Always	Always

C.121.4. 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.121.5. IDL Operation

```

```

C.121.6. Exceptions

This instruction does not generate synchronous exceptions.

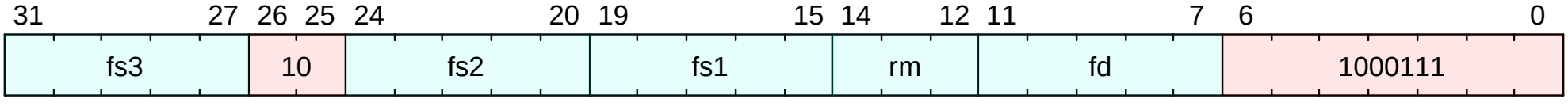
C.122. fmsub.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.122.1. Encoding



C.122.2. Description

No description available.

C.122.3. Access

M	S	U
Always	Always	Always

C.122.4. 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.122.5. IDL Operation

```

```

C.122.6. Exceptions

This instruction does not generate synchronous exceptions.

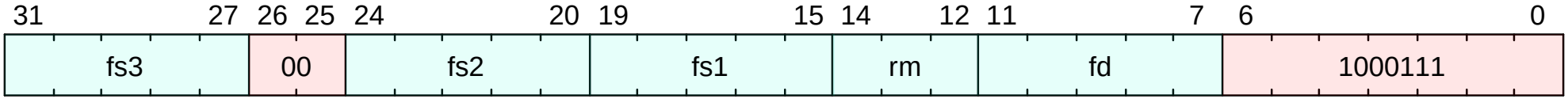
C.123. fmsub.s

Floating-Point Multiply-Subtract Single-Precision

This instruction is defined by:

F

C.123.1. Encoding



C.123.2. Description

The fmsub.s multiplies the values in `fs1` and `fs2`, subtracts the value in `fs3`, and writes the final result to `fd`.

C.123.3. Access

M	S	U
Always	Always	Always

C.123.4. 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.123.5. IDL Operation

```

```

C.123.6. 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.123.7. Exceptions

This instruction does not generate synchronous exceptions.

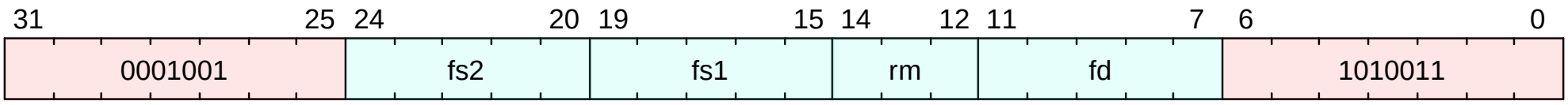
C.124. fmul.d

Floating-Point Multiply Double-Precision

This instruction is defined by:

(D ∪ Zdinx)

C.124.1. Encoding



C.124.2. Description

The fmul.d instruction performs the double-precision floating-point multiplication between fs1 and fs2. It is defined analogously to its single-precision counterpart, but operates on double-precision operands and produces double-precision results.

C.124.3. Access

M	S	U
Always	Always	Always

C.124.4. 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.124.5. IDL Operation

```

```

C.124.6. Exceptions

This instruction does not generate synchronous exceptions.

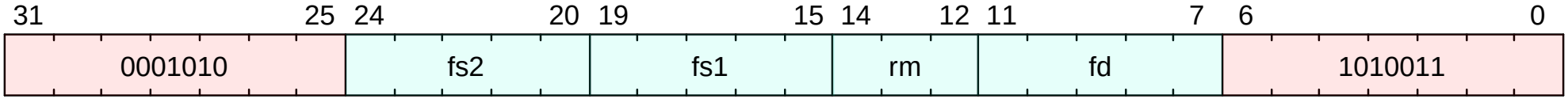
C.125. fmul.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.125.1. Encoding



C.125.2. Description

No description available.

C.125.3. Access

M	S	U
Always	Always	Always

C.125.4. Decode Variables

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

C.125.5. IDL Operation

```

```

C.125.6. Exceptions

This instruction does not generate synchronous exceptions.

C.126. fmul.s

Floating-Point Multiply Single-Precision

This instruction is defined by:

F

C.126.1. Encoding

C.126.2. Description

The fmul.s instruction performs the single-precision floating-point multiplication between `fs1` and `fs2`, and writes the result in `fd`.

C.126.3. Access

M	S	U
Always	Always	Always

C.126.4. Decode Variables

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

C.126.5. IDL Operation

```

```

C.126.6. Sail Operation

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

C.126.7. Exceptions

This instruction does not generate synchronous exceptions.

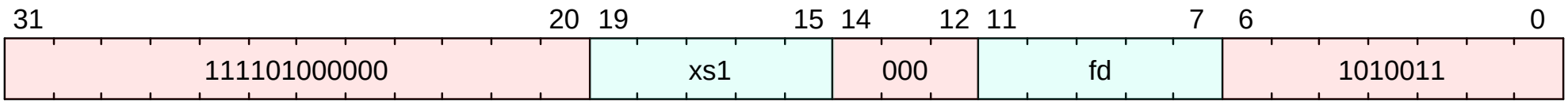
C.127. fmv.h.x

Half-precision floating-point move from integer

This instruction is defined by:

Zfhmin

C.127.1. Encoding



C.127.2. Description

Moves the half-precision value encoded in IEEE 754-2008 standard encoding from the lower 16 bits of integer register `xs1` to the floating-point register `fd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

C.127.3. Access

M	S	U
Always	Always	Always

C.127.4. Decode Variables

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

C.127.5. IDL Operation

```
check_f_ok($encoding);
Bits<16> hp_value = X[xs1][15:0];
f[fd] = nan_box<16, FLEN>(hp_value);
mark_f_state_dirty();
```

C.127.6. 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.127.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

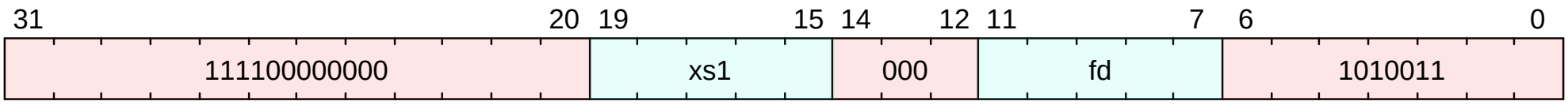
C.128. fmv.w.x

Floating-Point Move Single-Precision Word from Integer Register

This instruction is defined by:

F

C.128.1. Encoding



C.128.2. Description

The fmv.w.x instruction moves the single-precision value encoded in `IEEE 754-2008` standard encoding from the lower 32 bits of integer register `xs1` to the floating-point register `fd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

C.128.3. Access

M	S	U
Always	Always	Always

C.128.4. Decode Variables

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

C.128.5. IDL Operation

```
check_f_ok($encoding);
Bits<32> sp_value = X[xs1][31:0];
if (implemented?(ExtensionName::D)) {
  f[fd] = nan_box<32, 64>(sp_value);
} else {
  f[fd] = sp_value;
}
mark_f_state_dirty();
```

C.128.6. 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.128.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

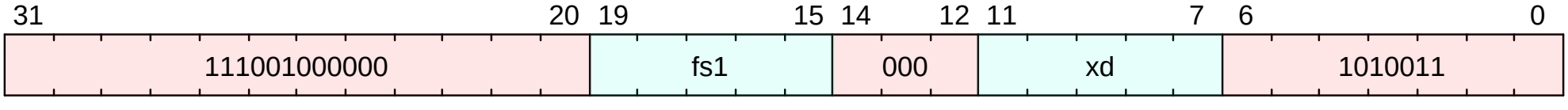
C.129. fmv.x.h

Move half-precision value from floating-point to integer register

This instruction is defined by:

Zfhmin

C.129.1. Encoding



C.129.2. Description

Moves the half-precision value in floating-point register fs1 represented in IEEE 754-2008 encoding to the lower 16 bits of integer register xd.

The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved.

The highest XLEN-16 bits of the destination register are filled with copies of the floating-point number's sign bit.

C.129.3. Access

M	S	U
Always	Always	Always

C.129.4. Decode Variables

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

C.129.5. IDL Operation

```
check_f_ok($encoding);
X[xd] = sext(f[fs1][15:0], 16);
```

C.129.6. 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.129.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

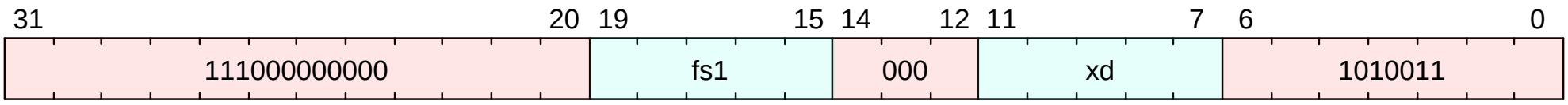
C.130. fmv.x.w

Floating-Point Move Single-Precision Word to Integer Register

This instruction is defined by:

F

C.130.1. Encoding



C.130.2. Description

The fmv.x.w instruction moves the single-precision value in floating-point register `fs1` represented in `IEEE 754-2008` encoding to the lower 32 bits of integer register `xd`. The bits are not modified in the transfer, and in particular, the payloads of non-canonical NaNs are preserved. For RV64, the higher 32 bits of the destination register are filled with copies of the floating-point number's sign bit.

C.130.3. Access

M	S	U
Always	Always	Always

C.130.4. Decode Variables

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

C.130.5. IDL Operation

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

C.130.6. 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.130.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

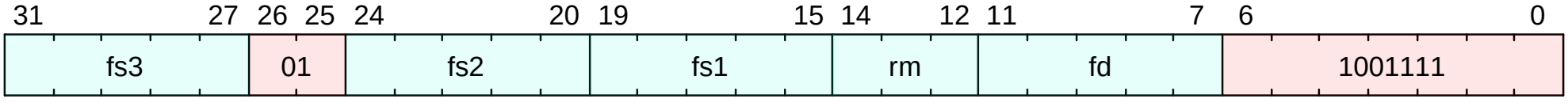
C.131. fnmadd.d

Floating-Point Negate-Multiply-Add Double-Precision

This instruction is defined by:

(D ⬜ Zdinx)

C.131.1. Encoding



C.131.2. Description

The fnmadd.d instruction multiplies the values in fs1 and fs2, negates the product, subtracts the value in fs3, and writes the final result to fd.

C.131.3. Access

M	S	U
Always	Always	Always

C.131.4. 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.131.5. IDL Operation

```

```

C.131.6. Exceptions

This instruction does not generate synchronous exceptions.

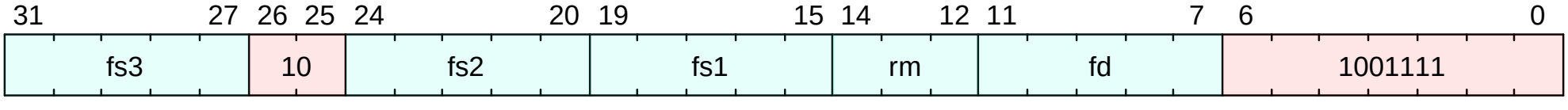
C.132. fnmadd.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.132.1. Encoding



C.132.2. Description

No description available.

C.132.3. Access

M	S	U
Always	Always	Always

C.132.4. 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.132.5. IDL Operation

```

```

C.132.6. Exceptions

This instruction does not generate synchronous exceptions.

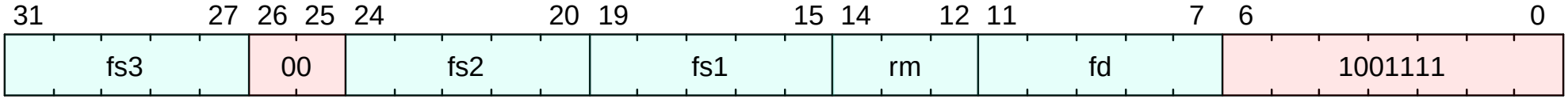
C.133. fnmadd.s

Floating-Point Negate-Multiply-Add Single-Precision

This instruction is defined by:

F

C.133.1. Encoding



C.133.2. Description

The fnmadd.s multiplies the values in `fs1` and `fs2`, negates the product, subtracts the value in `fs3`, and writes the final result to `fd`.

C.133.3. Access

M	S	U
Always	Always	Always

C.133.4. 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.133.5. IDL Operation

```

```

C.133.6. 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.133.7. Exceptions

This instruction does not generate synchronous exceptions.

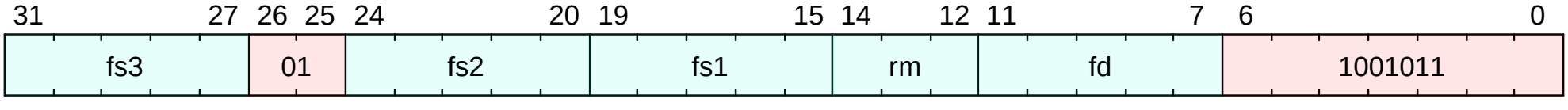
C.134. fnmsub.d

Floating-Point Negate-Multiply-Subtract Double-Precision

This instruction is defined by:

(D ⬜ Zdinx)

C.134.1. Encoding



C.134.2. Description

The fnmsub.d instruction multiplies the values in fs1 and fs2, negates the product, adds the value in fs3, and writes the final result to fd.

C.134.3. Access

M	S	U
Always	Always	Always

C.134.4. 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.134.5. IDL Operation

```

```

C.134.6. Exceptions

This instruction does not generate synchronous exceptions.

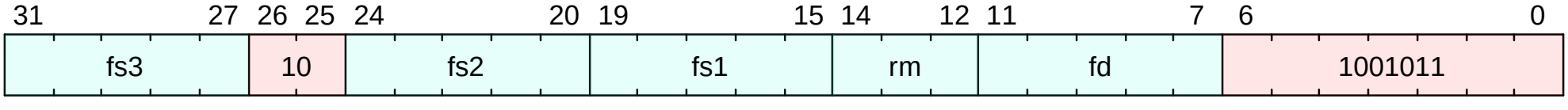
C.135. fnmsub.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.135.1. Encoding



C.135.2. Description

No description available.

C.135.3. Access

M	S	U
Always	Always	Always

C.135.4. 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.135.5. IDL Operation

```

```

C.135.6. Exceptions

This instruction does not generate synchronous exceptions.

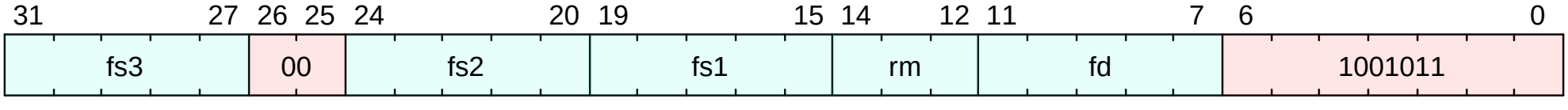
C.136. fnmsub.s

Floating-Point Negate-Multiply-Subtract Single-Precision

This instruction is defined by:

F

C.136.1. Encoding



C.136.2. Description

The fnmsub.s instruction multiplies the values in `fs1` and `fs2`, negates the product, adds the value in `fs3`, and writes the final result to `fd`.

C.136.3. Access

M	S	U
Always	Always	Always

C.136.4. 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.136.5. IDL Operation

```

```

C.136.6. 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.136.7. Exceptions

This instruction does not generate synchronous exceptions.

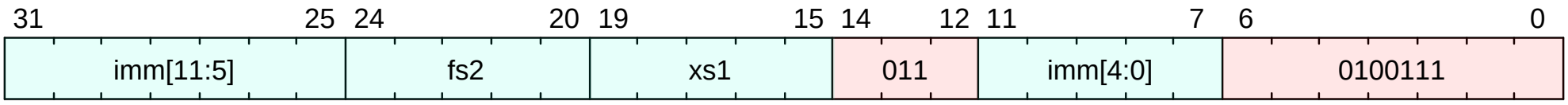
C.137. fsd

Floating-Point Store Double-Precision

This instruction is defined by:

D

C.137.1. Encoding



C.137.2. Description

The fsd instruction stores a double-precision value from the floating-point registers to memory. It is guaranteed to execute atomically if the effective address is naturally aligned and XLEN≥64. It doesn't modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

C.137.3. Access

M	S	U
Always	Always	Always

C.137.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

C.137.5. IDL Operation

```

```

C.137.6. Exceptions

This instruction does not generate synchronous exceptions.

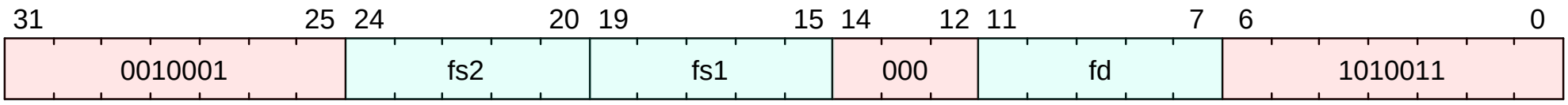
C.138. fsgnj.d

Floating-Point Sign-Inject Double-Precision

This instruction is defined by:

(D Zdinx)

C.138.1. Encoding



C.138.2. Description

The fsgnj.d instruction produces a result that takes all bits except the sign bit from fs1. The result's sign bit is taken from fs2s sign bit, and the result is written to the destination register `fd. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.138.3. Access

M	S	U
Always	Always	Always

C.138.4. Decode Variables

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

C.138.5. IDL Operation

```

```

C.138.6. Exceptions

This instruction does not generate synchronous exceptions.

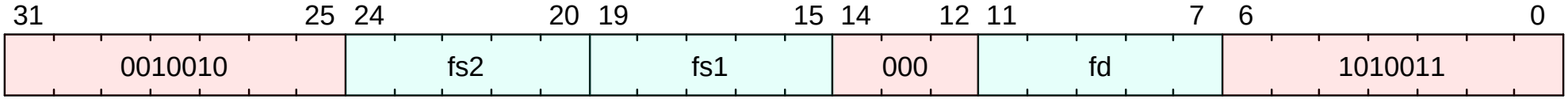
C.139. fsgnj.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.139.1. Encoding



C.139.2. Description

No description available.

C.139.3. Access

M	S	U
Always	Always	Always

C.139.4. Decode Variables

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

C.139.5. IDL Operation

```

```

C.139.6. Exceptions

This instruction does not generate synchronous exceptions.

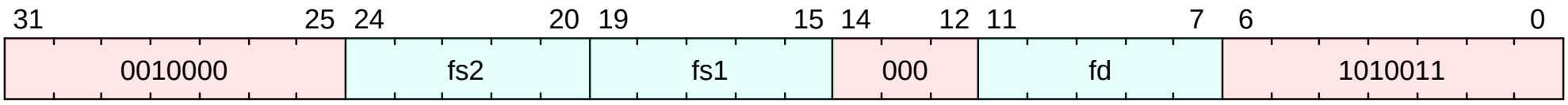
C.140. fsgnj.s

Floating-Point Sign-Inject Single-Precision

This instruction is defined by:

F

C.140.1. Encoding



C.140.2. Description

The fsgnj.s instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is taken from `fs2`'s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.140.3. Access

M	S	U
Always	Always	Always

C.140.4. Decode Variables

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

C.140.5. 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.140.6. 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.140.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

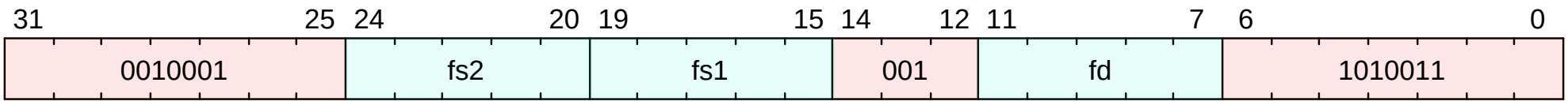
C.141. fsgnjn.d

Floating-Point Sign-Inject Negate Double-Precision

This instruction is defined by:

(D Zdinx)

C.141.1. Encoding



C.141.2. Description

The fsgnjn.d instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is opposite of `fs2s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.141.3. Access

M	S	U
Always	Always	Always

C.141.4. Decode Variables

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

C.141.5. IDL Operation

```

```

C.141.6. Exceptions

This instruction does not generate synchronous exceptions.

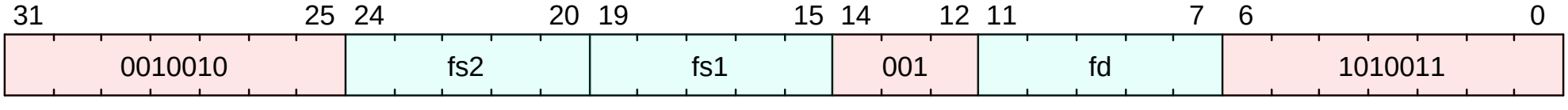
C.142. fsgnjn.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.142.1. Encoding



C.142.2. Description

No description available.

C.142.3. Access

M	S	U
Always	Always	Always

C.142.4. Decode Variables

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

C.142.5. IDL Operation

```

```

C.142.6. Exceptions

This instruction does not generate synchronous exceptions.

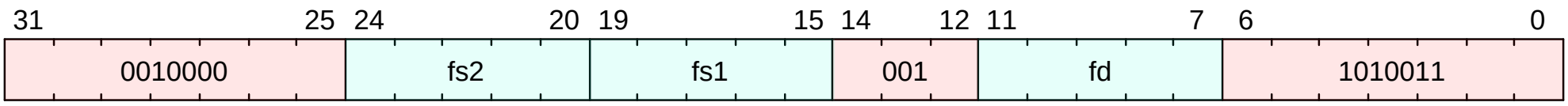
C.143. fsgnjn.s

Floating-Point Sign-Inject Negate Single-Precision

This instruction is defined by:

F

C.143.1. Encoding



C.143.2. Description

The fsgnjn.s instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is opposite of `fs2`'s sign bit, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.143.3. Access

M	S	U
Always	Always	Always

C.143.4. Decode Variables

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

C.143.5. 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.143.6. 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.143.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

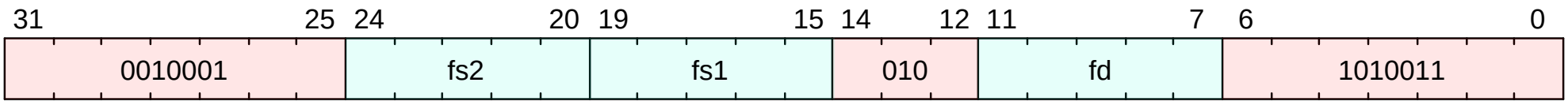
C.144. fsgnjx.d

Floating-Point Sign-Inject XOR Double-Precision

This instruction is defined by:

(D Zdinx)

C.144.1. Encoding



C.144.2. Description

The fsgnjx.d instruction produces a result that takes all bits except the sign bit from fs1. The result's sign bit is the XOR of sign bits of fs1 and fs2, and the result is written to the destination register fd. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.144.3. Access

M	S	U
Always	Always	Always

C.144.4. Decode Variables

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

C.144.5. IDL Operation

```

```

C.144.6. Exceptions

This instruction does not generate synchronous exceptions.

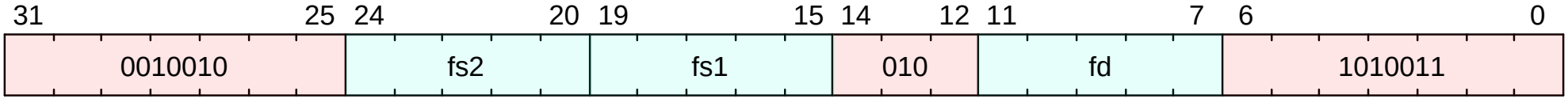
C.145. fsgnjx.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.145.1. Encoding



C.145.2. Description

No description available.

C.145.3. Access

M	S	U
Always	Always	Always

C.145.4. Decode Variables

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

C.145.5. IDL Operation

```

```

C.145.6. Exceptions

This instruction does not generate synchronous exceptions.

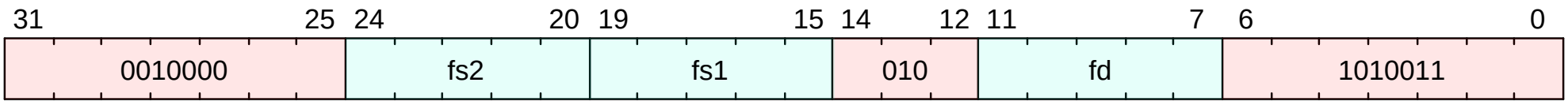
C.146. fsgnjx.s

Floating-Point Sign-Inject XOR Single-Precision

This instruction is defined by:

F

C.146.1. Encoding



C.146.2. Description

The fsgnjx.s instruction produces a result that takes all bits except the sign bit from `fs1`. The result's sign bit is the XOR of sign bits of `fs1` and `fs2`, and the result is written to the destination register `fd`. Sign-injection instructions do not set floating-point exception flags, nor do they canonicalize NaNs.

C.146.3. Access

M	S	U
Always	Always	Always

C.146.4. Decode Variables

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

C.146.5. 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.146.6. 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.146.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

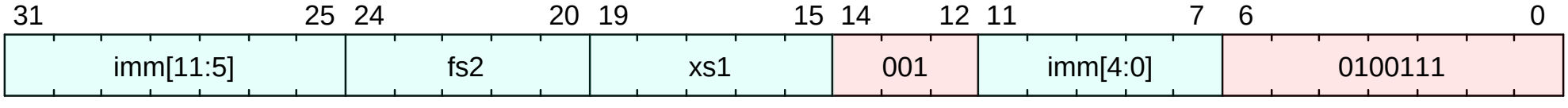
C.147. fsh

Half-precision floating-point store

This instruction is defined by:

Zfhmin

C.147.1. Encoding



C.147.2. Description

The fsh instruction stores a half-precision floating-point value from register *xd* to memory at address *xs1* + *imm*.

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

fsh ignores all but the lower 16 bits in *fs2*.

fsh is only guaranteed to execute atomically if the effective address is naturally aligned.

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> xs1 = $encoding[19:15];
Bits<5> fs2 = $encoding[24:20];
```

C.147.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[xs1] + $signed(imm);
Bits<16> hp_value = f[fs2][15:0];
write_memory<16>(virtual_address, hp_value, $encoding);
```

C.147.6. 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.147.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

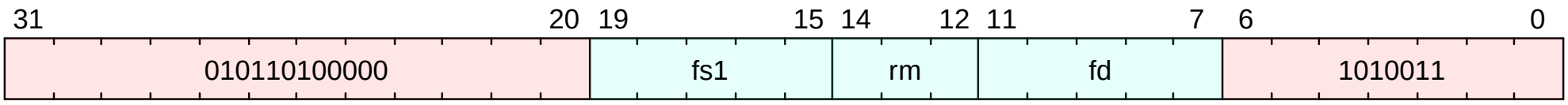
C.148. fsqrt.d

Floating-Point Square Root Double-Precision

This instruction is defined by:

(D Zdinx)

C.148.1. Encoding



C.148.2. Description

The fsqrt.d instruction computes the square root of fs1 and result is written in fd. It is defined analogously to its single-precision counterpart, but operates on double-precision operands and produces double-precision results.

C.148.3. Access

M	S	U
Always	Always	Always

C.148.4. Decode Variables

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

C.148.5. IDL Operation

```

```

C.148.6. Exceptions

This instruction does not generate synchronous exceptions.

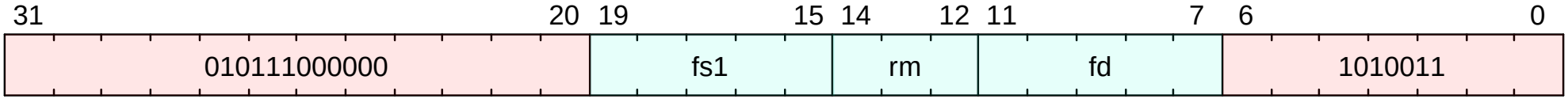
C.149. fsqrt.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.149.1. Encoding



C.149.2. Description

No description available.

C.149.3. Access

M	S	U
Always	Always	Always

C.149.4. Decode Variables

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

C.149.5. IDL Operation

```

```

C.149.6. Exceptions

This instruction does not generate synchronous exceptions.

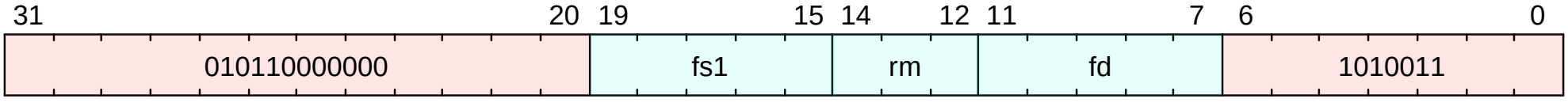
C.150. fsqrt.s

Floating-Point Square Root Single-Precision

This instruction is defined by:

F

C.150.1. Encoding



C.150.2. Description

The fsqrt.s instruction computes the square root of `fs1` and writes the result is written to `fd`.

C.150.3. Access

M	S	U
Always	Always	Always

C.150.4. Decode Variables

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

C.150.5. IDL Operation

```

```

C.150.6. 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.150.7. Exceptions

This instruction does not generate synchronous exceptions.

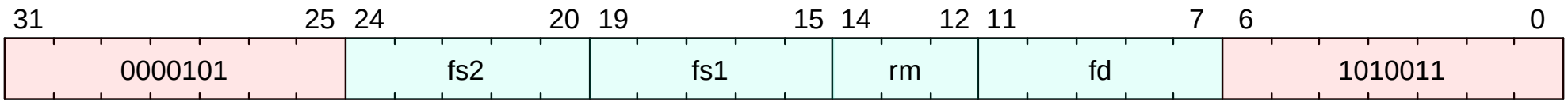
C.151. fsub.d

Floating-Point Subtract Double-Precision

This instruction is defined by:

(D ∥ Zdinx)

C.151.1. Encoding



C.151.2. Description

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

C.151.3. Access

M	S	U
Always	Always	Always

C.151.4. Decode Variables

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

C.151.5. IDL Operation

```

```

C.151.6. Exceptions

This instruction does not generate synchronous exceptions.

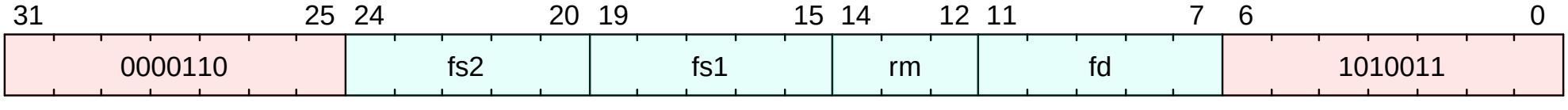
C.152. fsub.h

No synopsis available

This instruction is defined by:

(Zfh ▯ Zhinx)

C.152.1. Encoding



C.152.2. Description

No description available.

C.152.3. Access

M	S	U
Always	Always	Always

C.152.4. Decode Variables

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

C.152.5. IDL Operation

```

```

C.152.6. Exceptions

This instruction does not generate synchronous exceptions.

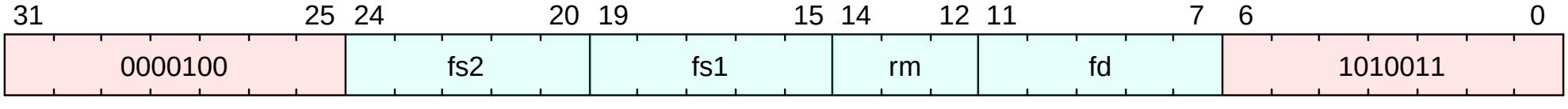
C.153. fsub.s

Floating-Point Subtract Single-Precision

This instruction is defined by:

F

C.153.1. Encoding



C.153.2. Description

The fsub.s instruction performs the single-precision floating-point subtraction of `fs2` from `fs1` and writes the result in `fd`.

C.153.3. Access

M	S	U
Always	Always	Always

C.153.4. 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.153.5. IDL Operation

```
check_f_ok($encoding);
RoundingMode mode = rm_to_mode(rm, $encoding);
f[fd] = f32_sub(f[fs1], f[fs2], mode);
```

C.153.6. 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.153.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

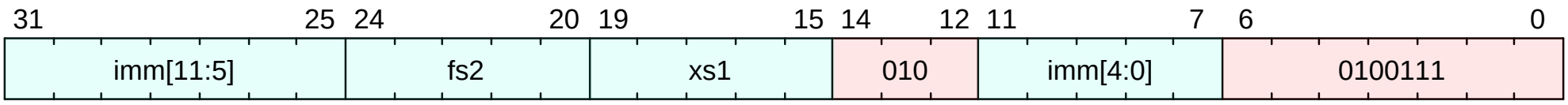
C.154. fsw

Floating-Point Store Single-Precision

This instruction is defined by:

F

C.154.1. Encoding



C.154.2. Description

The fsw instruction stores a single-precision floating-point value in `fs2` to memory at address `xs1` + `imm`. It does not modify the bits being transferred; in particular, the payloads of non-canonical NaNs are preserved.

C.154.3. Access

M	S	U
Always	Always	Always

C.154.4. Decode Variables

```
Bits<12> imm = {$encoding[31:25], $encoding[11:7]};
Bits<5> fs2 = $encoding[24:20];
Bits<5> xs1 = $encoding[19:15];
```

C.154.5. IDL Operation

```
check_f_ok($encoding);
XReg virtual_address = X[xs1] + $signed(imm);
write_memory<32>(virtual_address, f[fs2][31:0], $encoding);
```

C.154.6. 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.154.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

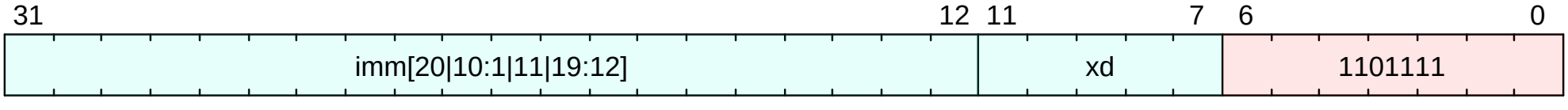
C.155. jal

Jump and link

This instruction is defined by:

I

C.155.1. Encoding



C.155.2. Description

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

C.155.3. Access

M	S	U
Always	Always	Always

C.155.4. 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.155.5. IDL Operation

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

C.155.6. 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.155.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

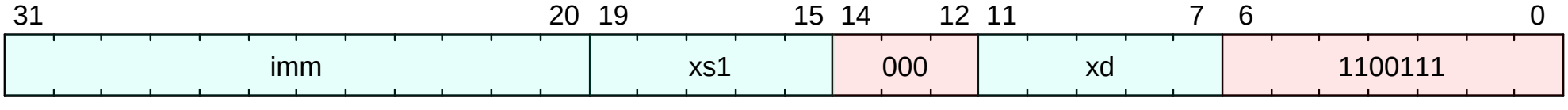
C.156. jalr

Jump and link register

This instruction is defined by:

I

C.156.1. Encoding



C.156.2. 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.156.3. Access

M	S	U
Always	Always	Always

C.156.4. Decode Variables

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

C.156.5. IDL Operation

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

C.156.6. 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.156.7. Exceptions

This instruction may result in the following synchronous exceptions:

- InstructionAddressMisaligned

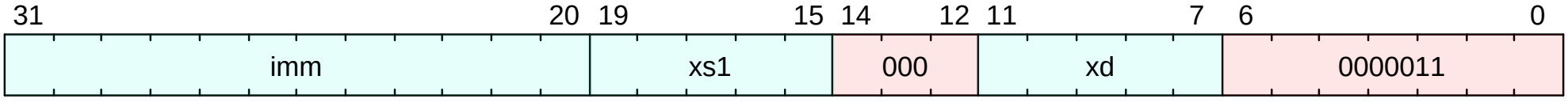
C.157. lb

Load byte

This instruction is defined by:

I

C.157.1. Encoding



C.157.2. 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.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

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

C.157.6. 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.157.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

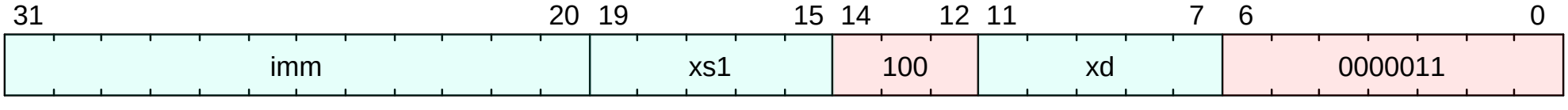
C.158. lbu

Load byte unsigned

This instruction is defined by:

I

C.158.1. Encoding



C.158.2. 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.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

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

C.158.6. 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.158.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

C.159. ld

Load doubleword

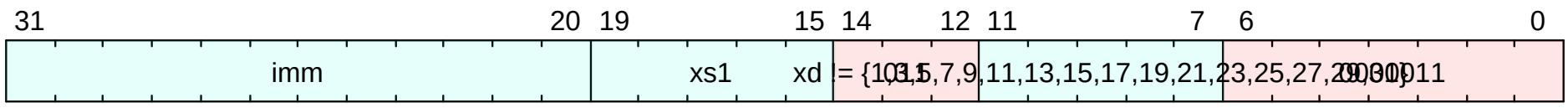
This instruction is defined by:

(I || Zilsd)

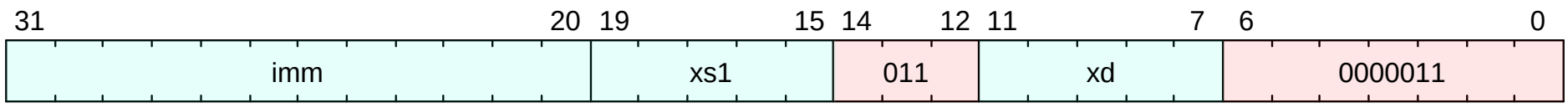
C.159.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.159.2. Description

For RV64, load 64 bits of data into register `xd` from an address formed by adding `xs1` to a signed offset.

<% if ext?(:Zilsd) %> For RV32, Loads a 64-bit value into registers xd and xd+1. The effective address is obtained by adding register xs1 to the sign-extended 12-bit offset. <% end %>

C.159.3. Access

M	S	U
Always	Always	Always

C.159.4. Decode Variables

RV32

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

RV64

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

C.159.5. IDL Operation

```
XReg virtual_address = X[xs1] + $signed(imm);
if (xlen() == 32) {
  if (implemented?(ExtensionName::Zilsd)) {
    Bits<64> data = read_memory<64>(virtual_address, $encoding);
    X[xd] = data[31:0];
    X[xd + 1] = data[63:32];
  } else {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else {
  X[xd] = read_memory<64>(virtual_address, $encoding);
}
```

C.159.6. 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.159.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- LoadAddressMisaligned
- LoadPageFault

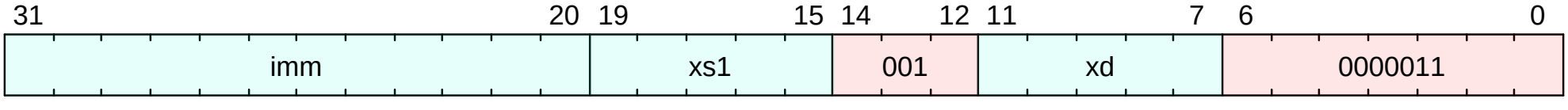
C.160. lh

Load halfword

This instruction is defined by:

I

C.160.1. Encoding



C.160.2. 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.160.3. Access

M	S	U
Always	Always	Always

C.160.4. Decode Variables

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

C.160.5. IDL Operation

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

C.160.6. 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.160.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

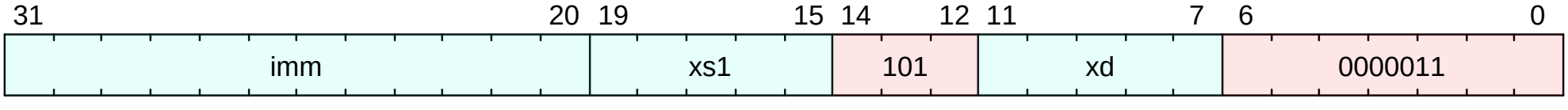
C.161. lhu

Load halfword unsigned

This instruction is defined by:

I

C.161.1. Encoding



C.161.2. 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.161.3. Access

M	S	U
Always	Always	Always

C.161.4. Decode Variables

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

C.161.5. IDL Operation

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

C.161.6. 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.161.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

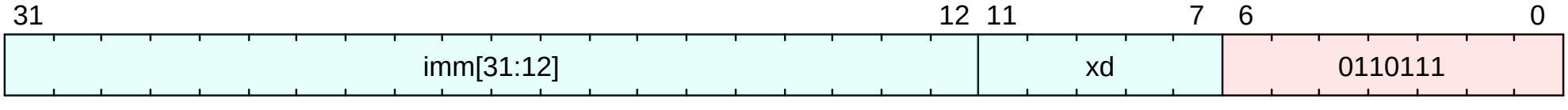
C.162. lui

Load upper immediate

This instruction is defined by:

I

C.162.1. Encoding



C.162.2. Description

Load the zero-extended imm into xd.

C.162.3. Access

M	S	U
Always	Always	Always

C.162.4. Decode Variables

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

C.162.5. IDL Operation

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

C.162.6. 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.162.7. Exceptions

This instruction does not generate synchronous exceptions.

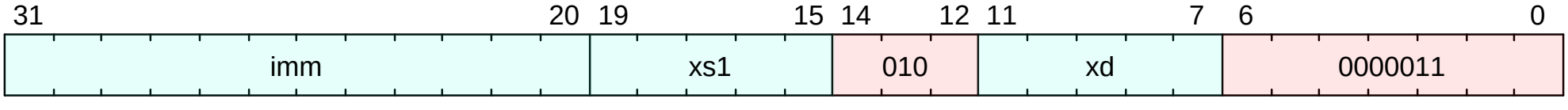
C.163. lw

Load word

This instruction is defined by:

I

C.163.1. Encoding



C.163.2. 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.163.3. Access

M	S	U
Always	Always	Always

C.163.4. Decode Variables

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

C.163.5. IDL Operation

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

C.163.6. 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.163.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault

- LoadAddressMisaligned
- LoadPageFault

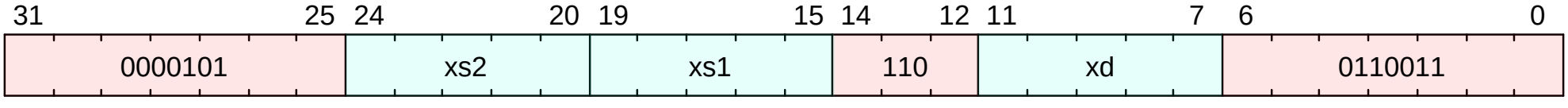
C.164. max

Maximum

This instruction is defined by:

Zbb

C.164.1. Encoding



C.164.2. Description

Returns the larger of two signed integers.

Software Hint

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

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

C.164.6. Sail Operation

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

C.164.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

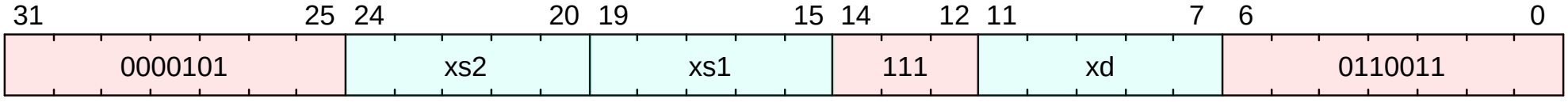
C.165. maxu

Unsigned maximum

This instruction is defined by:

Zbb

C.165.1. Encoding



C.165.2. Description

Returns the larger of two unsigned integers.

C.165.3. Access

M	S	U
Always	Always	Always

C.165.4. Decode Variables

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

C.165.5. IDL Operation

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

C.165.6. Sail Operation

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

C.165.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

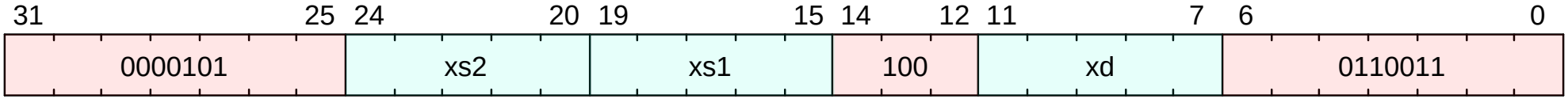
C.166. min

Minimum

This instruction is defined by:

Zbb

C.166.1. Encoding



C.166.2. Description

Returns the smaller of two signed integers.

C.166.3. Access

M	S	U
Always	Always	Always

C.166.4. Decode Variables

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

C.166.5. IDL Operation

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

C.166.6. Sail Operation

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

C.166.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

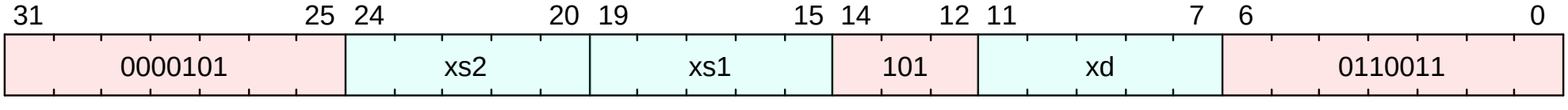
C.167. minu

Unsigned minimum

This instruction is defined by:

Zbb

C.167.1. Encoding



C.167.2. Description

Returns the smaller of two unsigned integers.

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

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

C.167.6. Sail Operation

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

C.167.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

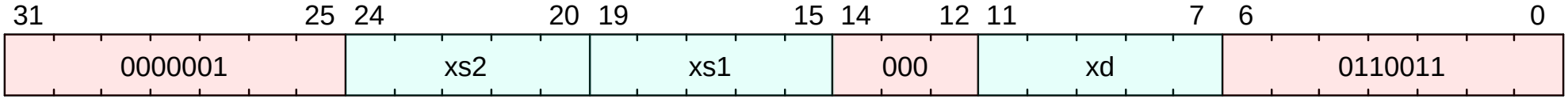
C.168. mul

Signed multiply

This instruction is defined by:

(M || Zmmul)

C.168.1. Encoding



C.168.2. Description

MUL performs an XLEN-bitxXLEN-bit multiplication of xs1 by xs2 and places the lower XLEN bits in the destination register. Any overflow is thrown away.

If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] xdh, xs1, xs2; MUL xdl, xs1, xs2 (source register specifiers must be in same order and xdh cannot be the same as xs1 or xs2). Microarchitectures can then fuse these into a single multiply operation instead of performing two separate multiplies.

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

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

C.168.6. 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.168.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

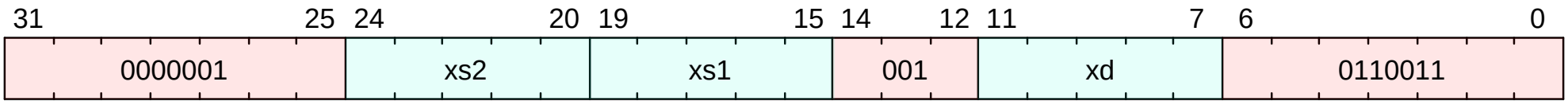
C.169. mulh

Signed multiply high

This instruction is defined by:

(M || Zmmul)

C.169.1. Encoding



C.169.2. Description

Multiply the signed values in xs1 to xs2, and store the upper half of the result in xd. The lower half is thrown away.

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

```
mulh xdh, xs1, xs2
mul  xdl, xs1, xs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

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

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<1> xs1_sign_bit = X[xs1][xlen() - 1];
Bits<MXLEN `* 2> src1 = {{xlen(){xs1_sign_bit}}, X[xs1]};
Bits<1> xs2_sign_bit = X[xs2][xlen() - 1];
Bits<MXLEN `* 2> src2 = {{xlen(){xs2_sign_bit}}, X[xs2]};
X[xd] = (src1 * src2)[(xlen() * 8'd2) - 1:xlen()];
```

C.169.6. 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.169.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

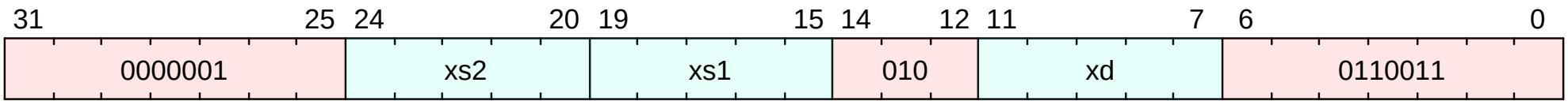
C.170. mulhsu

Signed/unsigned multiply high

This instruction is defined by:

(M || Zmmul)

C.170.1. Encoding



C.170.2. Description

Multiply the signed value in xs1 by the unsigned value in xs2, and store the upper half of the result in xd. The lower half is thrown away.

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

```
mulhsu xdh, xs1, xs2
mul    xdl, xs1, xs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

C.170.3. Access

M	S	U
Always	Always	Always

C.170.4. Decode Variables

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

C.170.5. IDL Operation

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<1> xs1_sign_bit = X[xs1][MXLEN - 1];
Bits<MXLEN * 8'd2> src1 = {{MXLEN{xs1_sign_bit}}, X[xs1]};
Bits<MXLEN * 8'd2> src2 = {{MXLEN{1'b0}}, X[xs2]};
X[xd] = (src1 * src2)[(MXLEN * 8'd2) - 1:MXLEN];
```

C.170.6. 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.170.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

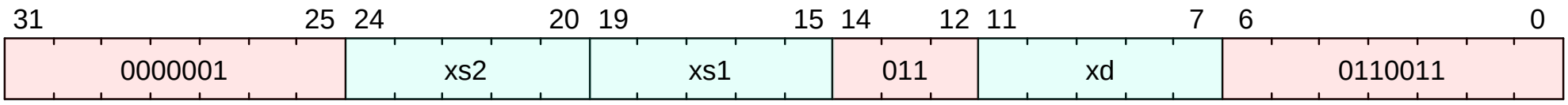
C.171. mulhu

Unsigned multiply high

This instruction is defined by:

(M || Zmmul)

C.171.1. Encoding



C.171.2. Description

Multiply the unsigned values in xs1 to xs2, and store the upper half of the result in xd. The lower half is thrown away.

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

```
mulhu xdh, xs1, xs2
mul   xdl, xs1, xs2
---
```

Microarchitectures may look for that sequence and fuse the operations.

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

```
if (implemented?(ExtensionName::M) && (CSR[misa].M == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
Bits<MXLEN * 8'd2> src1 = {{MXLEN{1'b0}}, X[xs1]};
Bits<MXLEN * 8'd2> src2 = {{MXLEN{1'b0}}, X[xs2]};
X[xd] = (src1 * src2)[(MXLEN * 8'd2) - 1:MXLEN];
```

C.171.6. 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.171.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

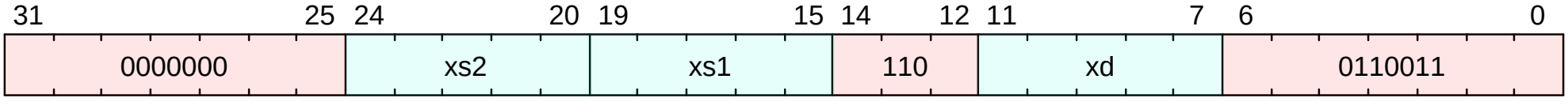
C.172. or

Or

This instruction is defined by:

I

C.172.1. Encoding



C.172.2. Description

Or xs1 with xs2, and store the result in xd

C.172.3. Access

M	S	U
Always	Always	Always

C.172.4. Decode Variables

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

C.172.5. IDL Operation

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

C.172.6. 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.172.7. Exceptions

This instruction does not generate synchronous exceptions.

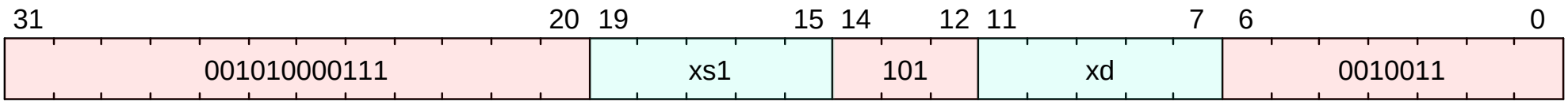
C.173. orc.b

Bitware OR-combine, byte granule

This instruction is defined by:

Zbb

C.173.1. Encoding



C.173.2. Description

Combines the bits within each byte using bitwise logical OR. This sets the bits of each byte in the result xd to all zeros if no bit within the respective byte of xs1 is set, or to all ones if any bit within the respective byte of xs1 is set.

C.173.3. Access

M	S	U
Always	Always	Always

C.173.4. Decode Variables

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

C.173.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg input = X[xs1];
XReg output = 0;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
  output[(i + 7):i] = (input[(i + 7):i] == 0) ? 8'd0 : ~8'd0;
}
X[xd] = output;
```

C.173.6. Sail Operation

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

C.173.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

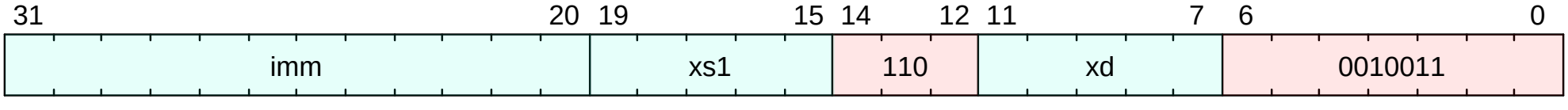
C.174. ori

Or immediate

This instruction is defined by:

I

C.174.1. Encoding



C.174.2. Description

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

C.174.3. Access

M	S	U
Always	Always	Always

C.174.4. Decode Variables

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

C.174.5. IDL Operation

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

C.174.6. 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.174.7. Exceptions

This instruction does not generate synchronous exceptions.

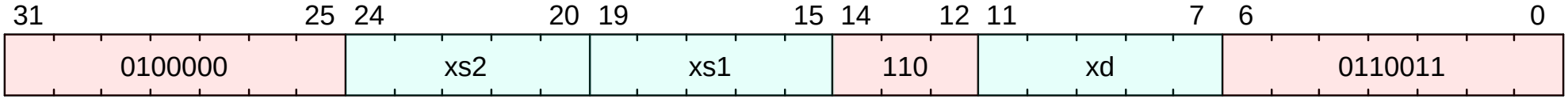
C.175. orn

OR with inverted operand

This instruction is defined by:

(Zbb || Zbkb)

C.175.1. Encoding



C.175.2. Description

Performs the bitwise logical OR operation between xs1 and the bitwise inversion of xs2.

C.175.3. Access

M	S	U
Always	Always	Always

C.175.4. Decode Variables

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

C.175.5. IDL Operation

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

C.175.6. Sail Operation

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

C.175.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

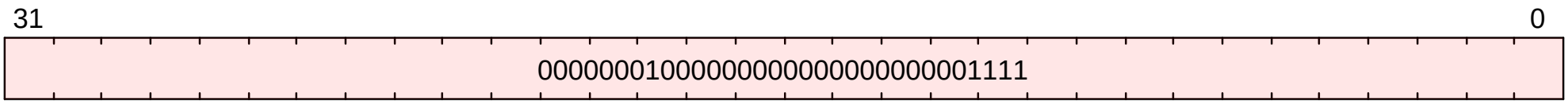
C.176. pause

Pause hint

This instruction is defined by:

Zihintpause

C.176.1. Encoding



C.176.2. Description

The PAUSE instruction is a HINT that indicates the current hart's rate of instruction retirement should be temporarily reduced or paused. The duration of its effect must be bounded and may be zero.

> Software can use the PAUSE instruction to reduce energy consumption while executing spin-wait code sequences. Multithreaded cores might temporarily relinquish execution resources to other harts when PAUSE is executed. It is recommended that a PAUSE instruction generally be included in the code sequence for a spin-wait loop.
>
> A future extension might add primitives similar to the x86 MONITOR/MWAIT instructions, which provide a more efficient mechanism to wait on writes to a specific memory location. However, these instructions would not supplant PAUSE. PAUSE is more appropriate when polling for non-memory events, when polling for multiple events, or when software does not know precisely what events it is polling for.
>
> The duration of a PAUSE instruction's effect may vary significantly within and among implementations. In typical implementations this duration should be much less than the time to perform a context switch, probably more on the rough order of an on-chip cache miss latency or a cacheless access to main memory.
>
> A series of PAUSE instructions can be used to create a cumulative delay loosely proportional to the number of PAUSE instructions. In spin-wait loops in portable code, however, only one PAUSE instruction should be used before re-evaluating loop conditions, else the hart might stall longer than optimal on some implementations, degrading system performance.

PAUSE is encoded as a FENCE instruction with *pred*=`W`, *succ*=`0`, *fm*=`0`, *xd*=`x0`, and *xs1*=`x0`.

> PAUSE is encoded as a hint within the FENCE opcode because some implementations are expected to deliberately stall the PAUSE instruction until outstanding memory transactions have completed. Because the successor set is null, however, PAUSE does not *mandate* any particular memory ordering—hence, it truly is a HINT.
>
> Like other FENCE instructions, PAUSE cannot be used within LR/SC sequences without voiding the forward-progress guarantee.
>
> The choice of a predecessor set of W is arbitrary, since the successor set is null. Other HINTs similar to PAUSE might be encoded with other predecessor sets.

C.176.3. Access

M	S	U
Always	Always	Always

C.176.4. Decode Variables

```
```

C.176.5. IDL Operation

```
pause();
```

C.176.6. Exceptions

This instruction does not generate synchronous exceptions.

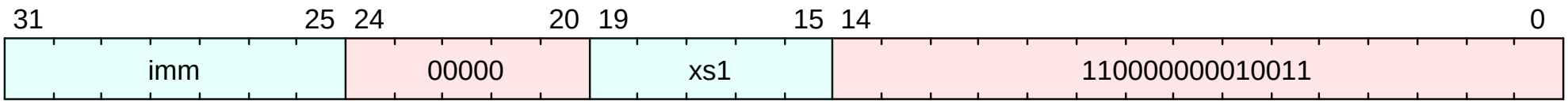
C.177. prefetch.i

Cache block prefetch for instruction fetch

This instruction is defined by:

Zicbop

C.177.1. Encoding



C.177.2. Description

A prefetch.i instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in xs1 and the sign-extended offset encoded in imm[11:0], where imm[4:0] equals 0b00000, is likely to be accessed by an instruction fetch in the near future.

C.177.3. Access

M	S	U
Always	Always	Always

C.177.4. Decode Variables

```
Bits<7> imm = $encoding[31:25];
Bits<5> xs1 = $encoding[19:15];
```

C.177.5. IDL Operation

```
XReg address = X[xs1] + $signed(imm << 5);
prefetch_instruction(address);
```

C.177.6. Exceptions

This instruction does not generate synchronous exceptions.

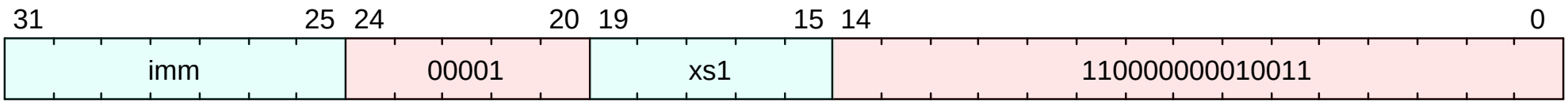
C.178. prefetch.r

Cache block prefetch for data read

This instruction is defined by:

Zicbop

C.178.1. Encoding



C.178.2. Description

A prefetch.r instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in xs1 and the sign-extended offset encoded in imm[11:0], where imm[4:0] equals 0b00000, is likely to be accessed by a data read (i.e. load) in the near future.

C.178.3. Access

M	S	U
Always	Always	Always

C.178.4. Decode Variables

```
Bits<7> imm = $encoding[31:25];
Bits<5> xs1 = $encoding[19:15];
```

C.178.5. IDL Operation

```
XReg address = X[xs1] + $signed(imm << 5);
prefetch_read(address);
```

C.178.6. Exceptions

This instruction does not generate synchronous exceptions.

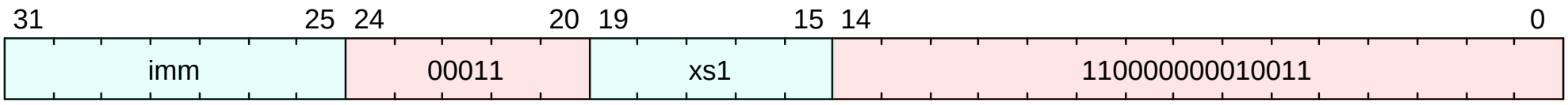
C.179. prefetch.w

Cache block prefetch for data write

This instruction is defined by:

Zicbop

C.179.1. Encoding



C.179.2. Description

A prefetch.w instruction indicates to hardware that the cache block whose effective address is the sum of the base address specified in xs1 and the sign-extended offset encoded in imm[11:0], where imm[4:0] equals 0b00000, is likely to be accessed by a data write (i.e. store) in the near future.

C.179.3. Access

M	S	U
Always	Always	Always

C.179.4. Decode Variables

```
Bits<7> imm = $encoding[31:25];
Bits<5> xs1 = $encoding[19:15];
```

C.179.5. IDL Operation

```
XReg address = X[xs1] + $signed(imm << 5);
prefetch_write(address);
```

C.179.6. Exceptions

This instruction does not generate synchronous exceptions.

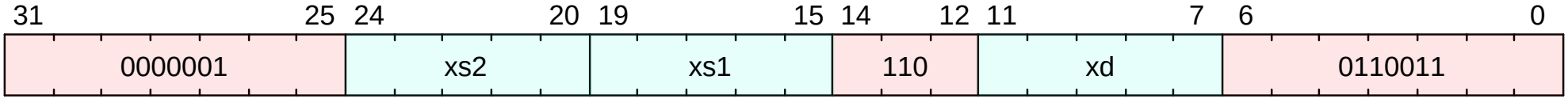
C.180. rem

Signed remainder

This instruction is defined by:

M

C.180.1. Encoding



C.180.2. Description

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

If the value in register xs2 is zero, write the value in xs1 into xd;

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

C.180.3. Access

M	S	U
Always	Always	Always

C.180.4. Decode Variables

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

C.180.5. IDL Operation

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

C.180.6. 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.180.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

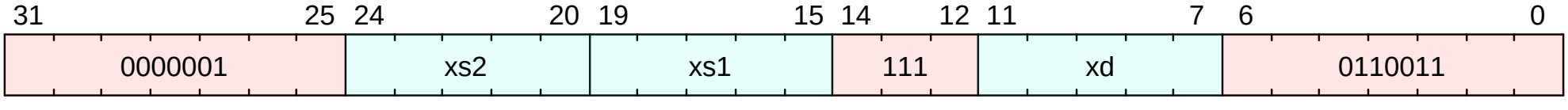
C.181. remu

Unsigned remainder

This instruction is defined by:

M

C.181.1. Encoding



C.181.2. Description

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

C.181.3. Access

M	S	U
Always	Always	Always

C.181.4. Decode Variables

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

C.181.5. IDL Operation

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

C.181.6. 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.181.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.182. rev8

Byte-reverse register (RV64 encoding)

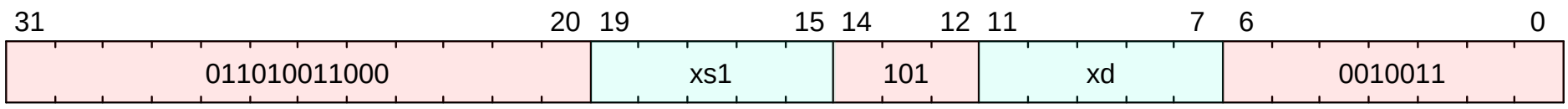
This instruction is defined by:

(Zbb || Zbkb)

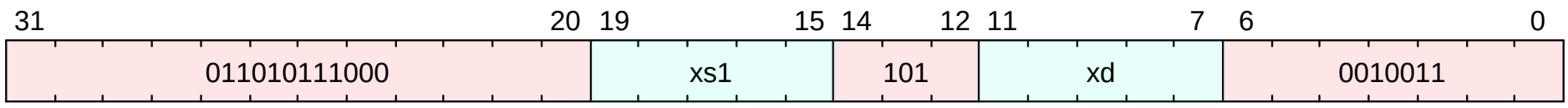
C.182.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.182.2. Description

Reverses the order of the bytes in rs1.

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

The byte-reverse operation is only available for the full register width. To emulate word-sized and halfword-sized byte-reversal, perform a `rev8 xd,xs1` followed by a `srai xd,xd,K`, where K is XLEN-32 and XLEN-16, respectively.

C.182.3. Access

M	S	U
Always	Always	Always

C.182.4. Decode Variables

RV32

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

RV64

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

C.182.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg input = X[xs1];
XReg output = 0;
XReg j = xlen() - 1;
for (U32 i = 0; i < (xlen() - 8); i = i + 8) {
  output[(i + 7):i] = input[j:(j - 7)];
  j = j - 8;
}
X[xd] = output;
```

C.182.6. Sail Operation

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

C.182.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

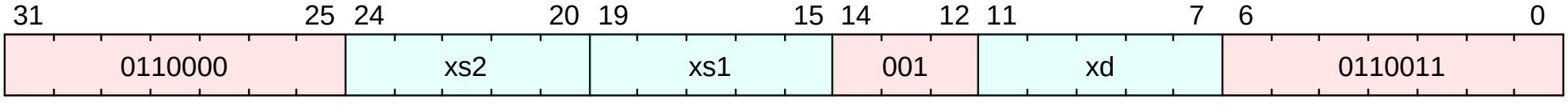
C.183. rol

Rotate left (Register)

This instruction is defined by:

(Zbb || Zbkb)

C.183.1. Encoding



C.183.2. Description

Performs a rotate left of xs1 by the amount in least-significant `log2(XLEN)` bits of xs2.

C.183.3. Access

M	S	U
Always	Always	Always

C.183.4. Decode Variables

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

C.183.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg shamt = (xlen() == 32) ? X[xs2][4:0] : X[xs2][5:0];
X[xd] = (X[xs1] << shamt) | (X[xs1] >> (xlen() - shamt));
```

C.183.6. Sail Operation

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

C.183.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

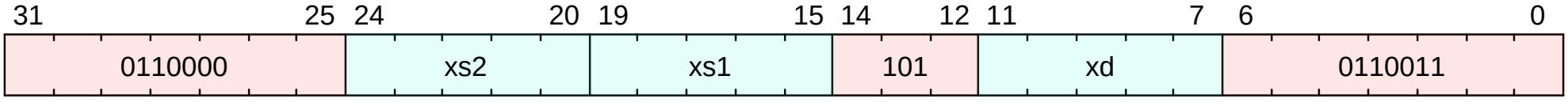
C.184. ror

Rotate right (Register)

This instruction is defined by:

(Zbb || Zbkb)

C.184.1. Encoding



C.184.2. Description

Performs a rotate right of xs1 by the amount in least-significant `log2(XLEN)` bits of xs2.

C.184.3. Access

M	S	U
Always	Always	Always

C.184.4. Decode Variables

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

C.184.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg shamt = (xlen() == 32) ? X[xs2][4:0] : X[xs2][5:0];
X[xd] = (X[xs1] >> shamt) | (X[xs1] << (xlen() - shamt));
```

C.184.6. Sail Operation

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

C.184.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

C.185. rori

Rotate right (Immediate)

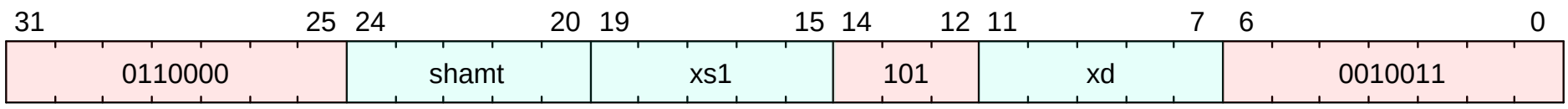
This instruction is defined by:

(Zbb || Zbkb)

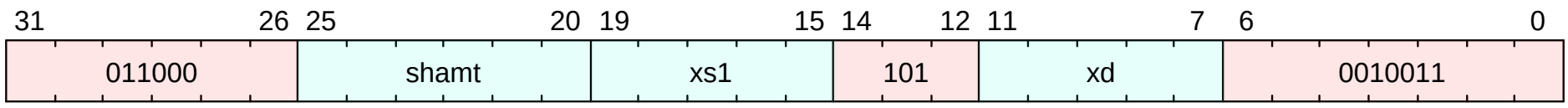
C.185.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.185.2. Description

Performs a rotate right of xs1 by the amount in the least-significant log2(XLEN) bits of shamt. For RV32, the encodings corresponding to shamt[5]=1 are reserved.

C.185.3. Access

M	S	U
Always	Always	Always

C.185.4. 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.185.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
XReg shamt = (xlen() == 32) ? shamt[4:0] : shamt[5:0];
X[xd] = (X[xs1] >> shamt) | (X[xs1] << (xlen() - shamt));
```

C.185.6. Sail Operation

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

C.185.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

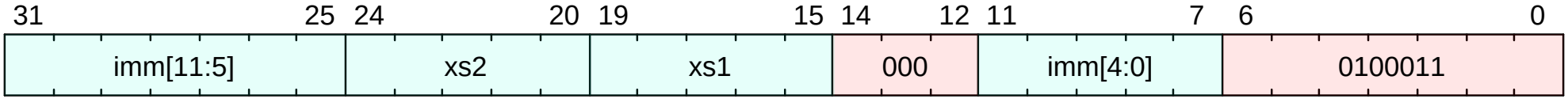
C.186. sb

Store byte

This instruction is defined by:

I

C.186.1. Encoding



C.186.2. Description

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

C.186.3. Access

M	S	U
Always	Always	Always

C.186.4. Decode Variables

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

C.186.5. IDL Operation

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

C.186.6. 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.186.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

C.187. sd

Store doubleword

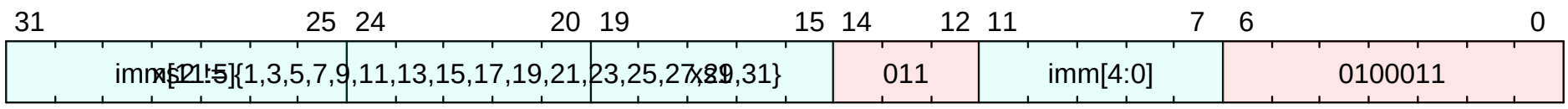
This instruction is defined by:

(I || Zilsd)

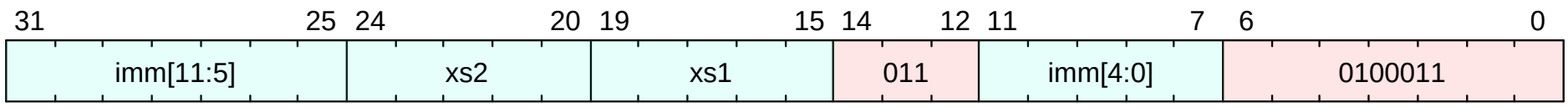
C.187.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.187.2. Description

For RV64, store 64 bits of data from register xs2 to an address formed by adding xs1 to a signed offset. <% if ext?(:Zilsd) %> For RV32, store doubleword from even/odd register pair. <% end %>

C.187.3. Access

M	S	U
Always	Always	Always

C.187.4. Decode Variables

RV32

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

RV64

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

C.187.5. IDL Operation

```
Bits<64> data;
XReg virtual_address = X[xs1] + $signed(imm);
if (xlen() == 32) {
  if (implemented?(ExtensionName::Zclsd)) {
    data = {X[xs2 + 1], X[xs2]};
  } else {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else {
  data = X[xs2];
}
write_memory<64>(virtual_address, data, $encoding);
```

C.187.6. Sail Operation

```
{
  let offset : xlenbits = sign_extend(imm);
```

```
/* Get the address, X(xs1) + offset.
   Some extensions perform additional checks on address validity. */
match ext_data_get_addr(xs1, offset, 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.187.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

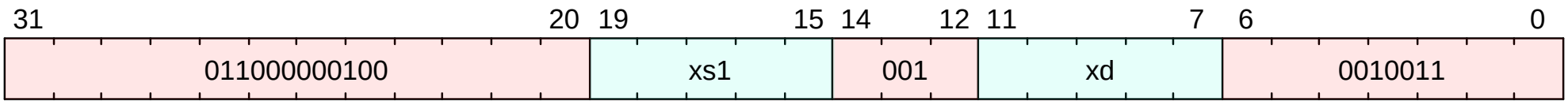
C.188. sext.b

Sign-extend byte

This instruction is defined by:

Zbb

C.188.1. Encoding



C.188.2. Description

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

C.188.3. Access

M	S	U
Always	Always	Always

C.188.4. Decode Variables

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

C.188.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xlen() == 32) {
  X[xd] = {{24{X[xs1][7]}}, X[xs1][7:0]};
} else if (xlen() == 64) {
  X[xd] = {{56{X[xs1][7]}}, X[xs1][7:0]};
}
```

C.188.6. Sail Operation

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

C.188.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

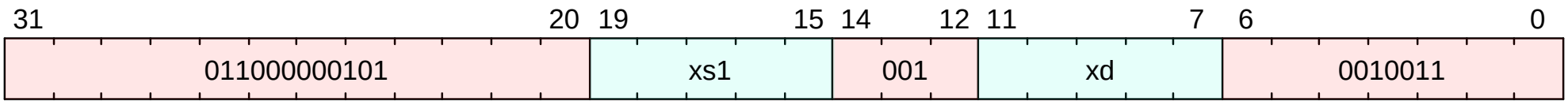
C.189. sext.h

Sign-extend halfword

This instruction is defined by:

Zbb

C.189.1. Encoding



C.189.2. Description

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

C.189.3. Access

M	S	U
Always	Always	Always

C.189.4. Decode Variables

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

C.189.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (xlen() == 32) {
  X[xd] = {{16{X[xs1][15]}}, X[xs1][15:0]};
} else if (xlen() == 64) {
  X[xd] = {{48{X[xs1][15]}}, X[xs1][15:0]};
}
```

C.189.6. Sail Operation

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

C.189.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

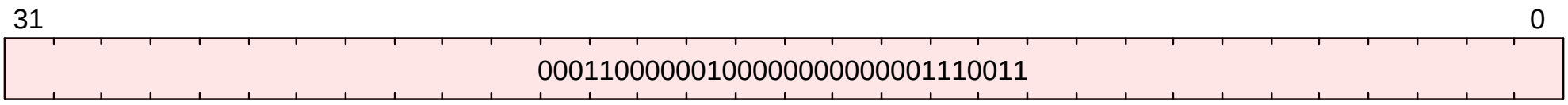
C.190. sfence.inval.ir

Order implicit page table reads after invalidation

This instruction is defined by:

Svinval

C.190.1. Encoding



C.190.2. Description

The sfence.inval.ir instruction guarantees that any previous sinval.vma instructions executed by the current hart are ordered before subsequent implicit references by that hart to the memory-management data structures.

C.190.3. Access

M	S	U
Always	Sometimes	Never

C.190.4. Decode Variables

```
```

C.190.5. IDL Operation

```
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
vma_type.global = true;
vma_type.smode = true;
if (CSR[misa].H == 1) {
  vma_type.vsmode = true;
  vma_type.gstage = true;
}
order_pgtbl_reads_after_vmafence(vma_type);
```

C.190.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

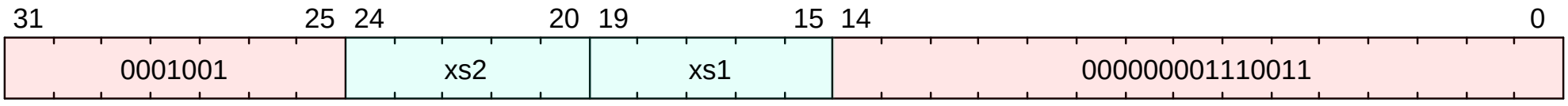
C.191. sfence.vma

Supervisor memory-management fence

This instruction is defined by:

S

C.191.1. Encoding



C.191.2. Description

The supervisor memory-management fence instruction `SFENCE.VMA` is used to synchronize updates to in-memory memory-management data structures with current execution. Instruction execution causes implicit reads and writes to these data structures; however, these implicit references are ordinarily not ordered with respect to explicit loads and stores. Executing an SFENCE.VMA instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before certain implicit references by subsequent instructions in that hart to the memory-management data structures. The specific set of operations ordered by SFENCE.VMA is determined by *xs1* and *xs2*, as described below. SFENCE.VMA is also used to invalidate entries in the address-translation cache associated with a hart (see [sv32algorithm]). Further details on the behavior of this instruction are described in [virt-control] and [pmp-vmem].

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

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

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

For the common case that the translation data structures have only been modified for a single address mapping (i.e., one page or superpage), *xs1* can specify a virtual address within that mapping to effect a translation fence for that mapping only. Furthermore, for the common case that the translation data structures have only been modified for a single address-space identifier, *xs2* can specify the address space. The behavior of SFENCE.VMA depends on *xs1* and *xs2* as follows:

- If *xs1*=`x0` and *xs2*=`x0`, the fence orders all reads and writes made to any level of the page tables, for all address spaces. The fence also invalidates all address-translation cache entries, for all address spaces.
- If *xs1*=`x0` and *xs2*≠`x0`, the fence orders all reads and writes made to any level of the page tables, but only for the address space identified by integer register *xs2*. Accesses to *global* mappings (see [translation]) are not ordered. The fence also invalidates all address-translation cache entries matching the address space identified by integer register *xs2*, except for entries containing global mappings.
- If *xs1*≠`x0` and *xs2*=`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *xs1*, for all address spaces. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *xs1*, for all address spaces.
- If *xs1*≠`x0` and *xs2*≠`x0`, the fence orders only reads and writes made to leaf page table entries corresponding to the virtual address in *xs1*, for the address space identified by integer register *xs2*. Accesses to global mappings are not ordered. The fence also invalidates all address-translation cache entries that contain leaf page table entries corresponding to the virtual address in *xs1* and that match the address space identified by integer register *xs2*, except for entries containing global mappings.

If the value held in *xs1* is not a valid virtual address, then the SFENCE.VMA instruction has no effect. No exception is raised in this case.

When *xs2*≠`x0`, bits SXLEN-1:ASIDMAX of the value held in *xs2* are reserved for future standard use. Until their use is defined by a standard extension, they should be zeroed by software and ignored by current implementations. Furthermore, if ASIDLEN<ASIDMAX, the implementation shall ignore bits ASIDMAX-1:ASIDLEN of the value held in *xs2*.

It is always legal to over-fence, e.g., by fencing only based on a subset of the bits in *xs1* and/or *xs2*, and/or by simply treating all SFENCE.VMA instructions as having *xs1*=`x0` and/or *xs2*=`x0`. For example, simpler implementations can ignore the virtual address in *xs1* and the ASID value in *xs2* and always perform a global fence. The choice not to raise an exception when an invalid virtual address is held in *xs1* facilitates this type of simplification.

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

associated explicit accesses, SFENCE.VMA does not necessarily place prior explicit accesses before subsequent explicit accesses in the global memory order. These implicit loads also need not otherwise obey normal program order semantics with respect to prior loads or stores to the same address.

> A consequence of this specification is that an implementation may use any translation for an address that was valid at any time since the most recent SFENCE.VMA that subsumes that address. In particular, if a leaf PTE is modified but a subsuming SFENCE.VMA is not executed, either the old translation or the new translation will be used, but the choice is unpredictable. The behavior is otherwise well-defined.
>
> In a conventional TLB design, it is possible for multiple entries to match a single address if, for example, a page is upgraded to a superpage without first clearing the original non-leaf PTE's valid bit and executing an SFENCE.VMA with *xs1*=`x0`. In this case, a similar remark applies: it is unpredictable whether the old non-leaf PTE or the new leaf PTE is used, but the behavior is otherwise well defined.
>
> Another consequence of this specification is that it is generally unsafe to update a PTE using a set of stores of a width less than the width of the PTE, as it is legal for the implementation to read the PTE at any time, including when only some of the partial stores have taken effect.

> This specification permits the caching of PTEs whose V (Valid) bit is clear. Operating systems must be written to cope with this possibility, but implementers are reminded that eagerly caching invalid PTEs will reduce performance by causing additional page faults.

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

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

> The following common situations typically require executing an SFENCE.VMA instruction:
>
> - When software recycles an ASID (i.e., reassociates it with a different page table), it should *first* change satp to point to the new page table using the recycled ASID, *then* execute SFENCE.VMA with *xs1*=`x0` and *xs2* set to the recycled ASID. Alternatively, software can execute the same SFENCE.VMA instruction while a different ASID is loaded into satp, provided the next time satp is loaded with the recycled ASID, it is simultaneously loaded with the new page table.
> - If the implementation does not provide ASIDs, or software chooses to always use ASID 0, then after every satp write, software should execute SFENCE.VMA with *xs1*=`x0`. In the common case that no global translations have been modified, *xs2* should be set to a register other than `x0` but which contains the value zero, so that global translations are not flushed.
> - If software modifies a non-leaf PTE, it should execute SFENCE.VMA with *xs1*=`x0`. If any PTE along the traversal path had its G bit set, *xs2* must be `x0`; otherwise, *xs2* should be set to the ASID for which the translation is being modified.
> - If software modifies a leaf PTE, it should execute SFENCE.VMA with *xs1* set to a virtual address within the page. If any PTE along the traversal path had its G bit set, *xs2* must be `x0`; otherwise, *xs2* should be set to the ASID for which the translation is being modified.
> - For the special cases of increasing the permissions on a leaf PTE and changing an invalid PTE to a valid leaf, software may choose to execute the SFENCE.VMA lazily. After modifying the PTE but before executing SFENCE.VMA, either the new or old permissions will be used. In the latter case, a page-fault exception might occur, at which point software should execute SFENCE.VMA in accordance with the previous bullet point.

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

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

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

C.191.3. Access

M	S	U
Always	Always	Never

C.191.4. Decode Variables

```
Bits<5> xs2 = $encoding[24:20];
```

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

C.191.5. IDL Operation

```
XReg vaddr = X[xs1];
Bits<16> asid = X[xs2][ASID_WIDTH - 1:0];
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[mstatus].TVM == 1 && mode() == PrivilegeMode::S) || (mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && CSR[hstatus].VTVM == 1 && mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
if (!implemented?(ExtensionName::Sv32) && !implemented?(ExtensionName::Sv39) && !implemented?(ExtensionName::Sv48) && !
implemented?(ExtensionName::Sv57)) {
  if (TRAP_ON_SFENCE_VMA_WHEN_SATP_MODE_IS_READ_ONLY) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
VmaOrderType vma_type;
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VS) {
  vma_type.vsmode = true;
  vma_type.single_vmid = true;
  vma_type.vmid = CSR[hgatp].VMID;
} else {
  vma_type.smode = true;
}
if ((xs1 == 0) && (xs2 == 0)) {
  vma_type.global = true;
  order_pgtbl_writes_before_vmafence(vma_type);
  invalidate_translations(vma_type);
  order_pgtbl_reads_after_vmafence(vma_type);
} else if ((xs1 == 0) && (xs2 != 0)) {
  vma_type.single_asid = true;
  vma_type.asid = asid;
  order_pgtbl_writes_before_vmafence(vma_type);
  invalidate_translations(vma_type);
  order_pgtbl_reads_after_vmafence(vma_type);
} else if ((xs1 != 0) && (xs2 == 0)) {
  if (canonical_vaddr?(vaddr)) {
    vma_type.single_vaddr = true;
    vma_type.vaddr = vaddr;
    order_pgtbl_writes_before_vmafence(vma_type);
    invalidate_translations(vma_type);
    order_pgtbl_reads_after_vmafence(vma_type);
  }
} else {
  if (canonical_vaddr?(vaddr)) {
    vma_type.single_asid = true;
    vma_type.asid = asid;
    vma_type.single_vaddr = true;
    vma_type.vaddr = vaddr;
    order_pgtbl_writes_before_vmafence(vma_type);
    invalidate_translations(vma_type);
    order_pgtbl_reads_after_vmafence(vma_type);
  }
}
```

C.191.6. Sail Operation

```
{
  let addr : option(xlenbits) = if rs1 == 0b00000 then None() else Some(X(rs1));
  let asid : option(xlenbits) = if rs2 == 0b00000 then None() else Some(X(rs2));
  match cur_privilege {
    User       => { handle_illegal(); RETIRE_FAIL },
    Supervisor => match (architecture(get_mstatus_SXL(mstatus)), mstatus.TVM()) {
                    (Some(_), 0b1)  => { handle_illegal(); RETIRE_FAIL },
```

```
                    (Some(_), 0b0) => { flush_TLB(asid, addr); RETIRE_SUCCESS },
                    (_, _)         => internal_error(__FILE__, __LINE__, "unimplemented sfence architecture")
                  },
    Machine    => { flush_TLB(asid, addr); RETIRE_SUCCESS }
  }
}
```

C.191.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

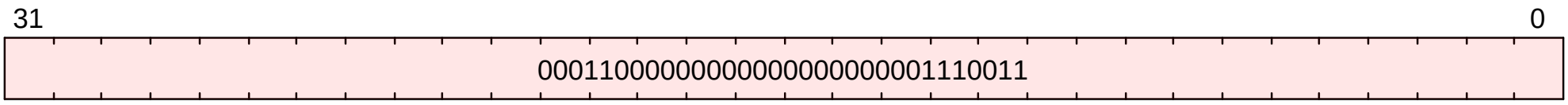
C.192. sfence.w.inval

Order writes before sfence

This instruction is defined by:

Svinval

C.192.1. Encoding



C.192.2. Description

The sfence.w.inval instruction guarantees that any previous stores already visible to the current RISC-V hart are ordered before subsequent sinval.vma instructions executed by the same hart.

C.192.3. Access

M	S	U
Always	Sometimes	Never

C.192.4. Decode Variables

```
```

C.192.5. IDL Operation

```
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
vma_type.global = true;
vma_type.smode = true;
if (CSR[misa].H == 1) {
  vma_type.vsmode = true;
  vma_type.gstage = true;
}
order_pgtbl_writes_before_vmafence(vma_type);
```

C.192.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

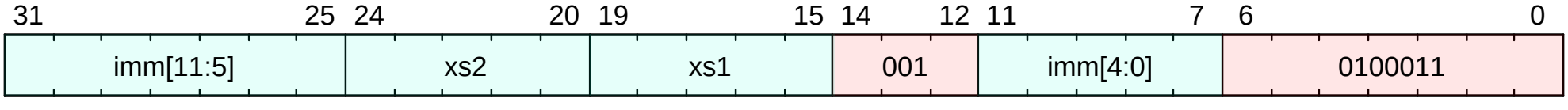
C.193. sh

Store halfword

This instruction is defined by:

I

C.193.1. Encoding



C.193.2. Description

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

C.193.3. Access

M	S	U
Always	Always	Always

C.193.4. Decode Variables

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

C.193.5. IDL Operation

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

C.193.6. 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.193.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

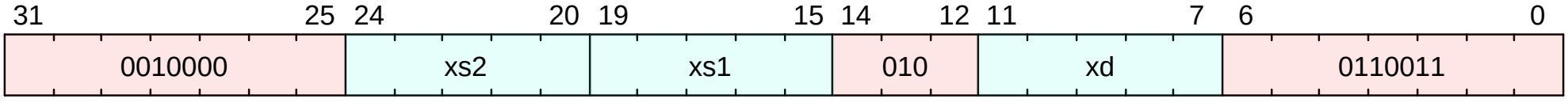
C.194. sh1add

Shift left by 1 and add

This instruction is defined by:

Zba

C.194.1. Encoding



C.194.2. Description

Shifts xs1 to the left by 1 bit and adds it to xs2.

C.194.3. Access

M	S	U
Always	Always	Always

C.194.4. Decode Variables

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

C.194.5. IDL Operation

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

C.194.6. Sail Operation

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

C.194.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

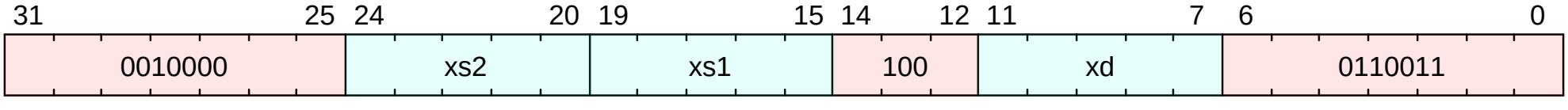
C.195. sh2add

Shift left by 2 and add

This instruction is defined by:

Zba

C.195.1. Encoding



C.195.2. Description

Shifts xs1 to the left by 2 places and adds it to xs2.

C.195.3. Access

M	S	U
Always	Always	Always

C.195.4. Decode Variables

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

C.195.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[xd] = X[xs2] + (X[xs1] << 2);
```

C.195.6. Sail Operation

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

C.195.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

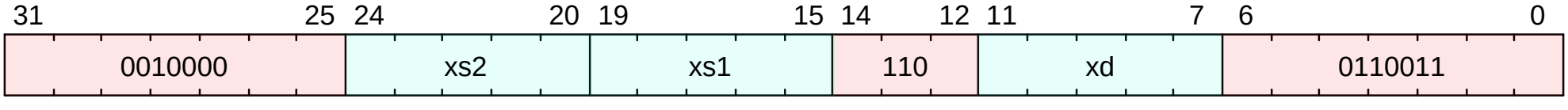
C.196. sh3add

Shift left by 3 and add

This instruction is defined by:

Zba

C.196.1. Encoding



C.196.2. Description

Shifts xs1 to the left by 3 places and adds it to xs2.

C.196.3. Access

M	S	U
Always	Always	Always

C.196.4. Decode Variables

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

C.196.5. IDL Operation

```
if (implemented?(ExtensionName::B) && (CSR[misa].B == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
X[xd] = X[xs2] + (X[xs1] << 3);
```

C.196.6. Sail Operation

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

C.196.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

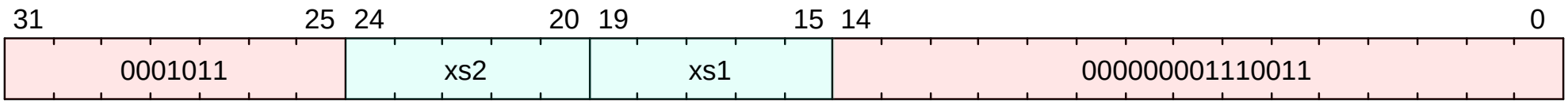
C.197. sinval.vma

Invalidate cached address translations

This instruction is defined by:

Svinval

C.197.1. Encoding



C.197.2. Description

The sinval.vma instruction invalidates any address-translation cache entries that an sfence.vma instruction with the same values of xs1 and xs2 would invalidate. However, unlike sfence.vma, sinval.vma instructions are only ordered with respect to sfence.vma, sfence.w.inval, and sfence.inval.ir instructions as defined below.

C.197.3. Access

M	S	U
Always	Sometimes	Never

C.197.4. Decode Variables

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

C.197.5. IDL Operation

```
XReg vaddr = X[xs1];
Bits<ASID_WIDTH> asid = X[xs2][ASID_WIDTH - 1:0];
if (CSR[mstatus].TVM == 1 && mode() == PrivilegeMode::S) || (mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && CSR[hstatus].VTVM == 1 && mode() == PrivilegeMode::VS) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
if (mode() == PrivilegeMode::U) {
  raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
}
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VU) {
  raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
}
VmaOrderType vma_type;
if (CSR[misa].H == 1 && mode() == PrivilegeMode::VS) {
  vma_type.vsmode = true;
  vma_type.single_vmid = true;
  vma_type.vmid = CSR[hgatp].VMID;
} else {
  vma_type.smode = true;
}
if ((xs1 == 0) && (xs2 == 0)) {
  vma_type.global = true;
  invalidate_translations(vma_type);
} else if ((xs1 == 0) && (xs2 != 0)) {
  vma_type.single_asid = true;
  vma_type.asid = asid;
  invalidate_translations(vma_type);
} else if ((xs1 != 0) && (xs2 == 0)) {
  if (canonical_vaddr?(vaddr)) {
    vma_type.single_vaddr = true;
    vma_type.vaddr = vaddr;
    invalidate_translations(vma_type);
  }
} else {
  if (canonical_vaddr?(vaddr)) {
    vma_type.single_asid = true;
```

```
    vma_type.asid = asid;
    vma_type.single_vaddr = true;
    vma_type.vaddr = vaddr;
    invalidate_translations(vma_type);
  }
}
```

C.197.6. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

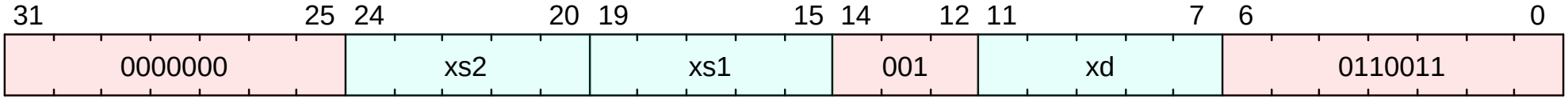
C.198. sll

Shift left logical

This instruction is defined by:

I

C.198.1. Encoding



C.198.2. Description

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

C.198.3. Access

M	S	U
Always	Always	Always

C.198.4. Decode Variables

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

C.198.5. IDL Operation

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

C.198.6. 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.198.7. Exceptions

This instruction does not generate synchronous exceptions.

C.199. slli

Shift left logical immediate

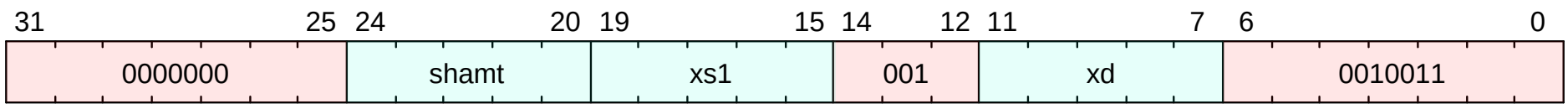
This instruction is defined by:

I

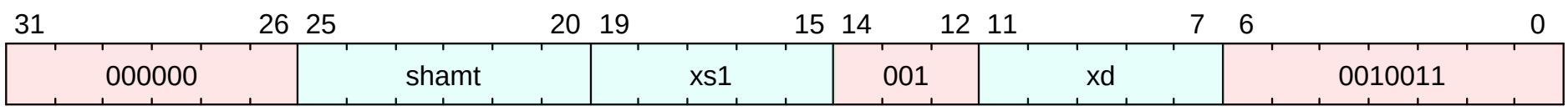
C.199.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.199.2. Description

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

C.199.3. Access

M	S	U
Always	Always	Always

C.199.4. 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.199.5. IDL Operation

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

C.199.6. 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.199.7. Exceptions

This instruction does not generate synchronous exceptions.

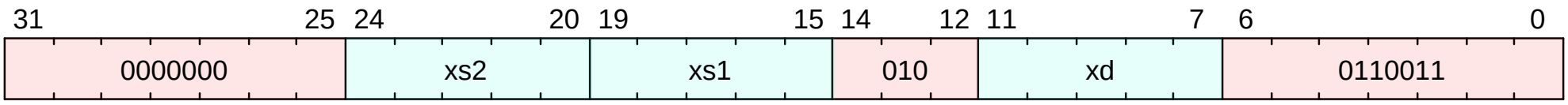
C.200. slt

Set on less than

This instruction is defined by:

I

C.200.1. Encoding



C.200.2. 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.200.3. Access

M	S	U
Always	Always	Always

C.200.4. Decode Variables

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

C.200.5. IDL Operation

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

C.200.6. 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.200.7. Exceptions

This instruction does not generate synchronous exceptions.

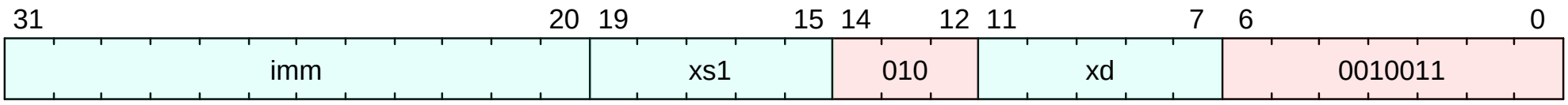
C.201. slti

Set on less than immediate

This instruction is defined by:

I

C.201.1. Encoding



C.201.2. 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.201.3. Access

M	S	U
Always	Always	Always

C.201.4. Decode Variables

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

C.201.5. IDL Operation

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

C.201.6. 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.201.7. Exceptions

This instruction does not generate synchronous exceptions.

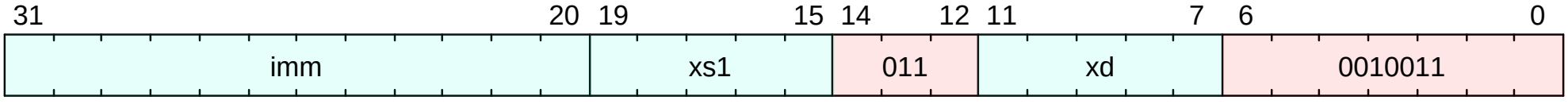
C.202. sltiu

Set on less than immediate unsigned

This instruction is defined by:

I

C.202.1. Encoding



C.202.2. 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.202.3. Access

M	S	U
Always	Always	Always

C.202.4. Decode Variables

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

C.202.5. IDL Operation

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

C.202.6. 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.202.7. Exceptions

This instruction does not generate synchronous exceptions.

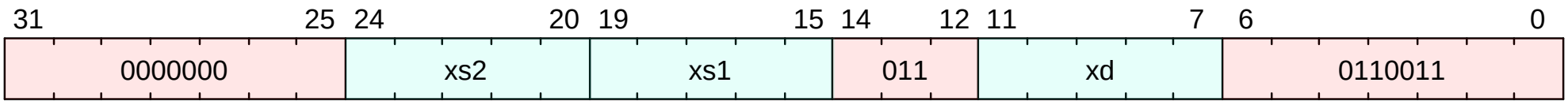
C.203. sltu

Set on less than unsigned

This instruction is defined by:

I

C.203.1. Encoding



C.203.2. 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.203.3. Access

M	S	U
Always	Always	Always

C.203.4. Decode Variables

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

C.203.5. IDL Operation

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

C.203.6. 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.203.7. Exceptions

This instruction does not generate synchronous exceptions.

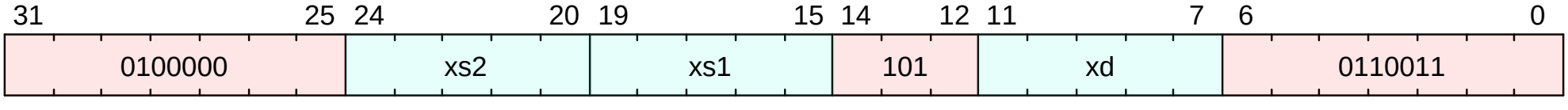
C.204. sra

Shift right arithmetic

This instruction is defined by:

I

C.204.1. Encoding



C.204.2. 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.204.3. Access

M	S	U
Always	Always	Always

C.204.4. Decode Variables

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

C.204.5. IDL Operation

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

C.204.6. 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.204.7. Exceptions

This instruction does not generate synchronous exceptions.

C.205. srai

Shift right arithmetic immediate

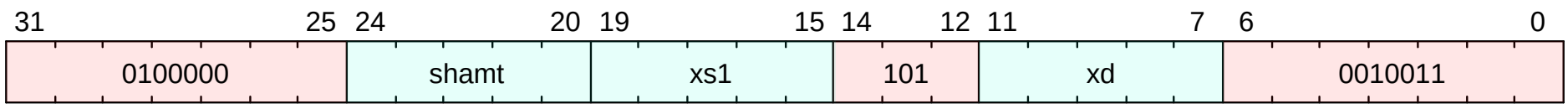
This instruction is defined by:

I

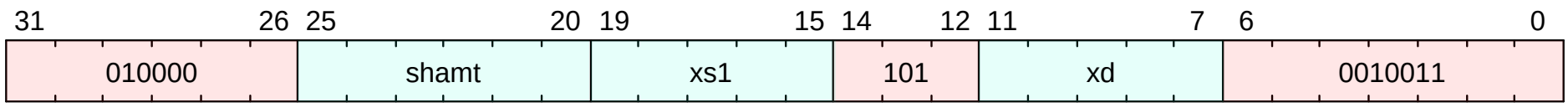
C.205.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.205.2. 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.205.3. Access

M	S	U
Always	Always	Always

C.205.4. 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.205.5. IDL Operation

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

C.205.6. 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.205.7. Exceptions

This instruction does not generate synchronous exceptions.

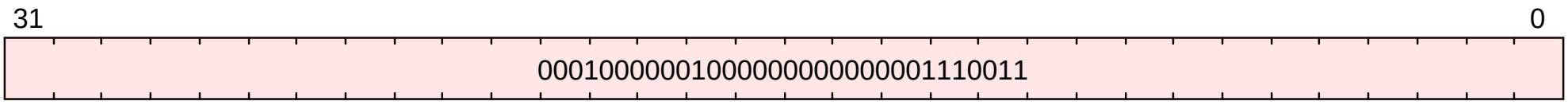
C.206. sret

Supervisor Mode Return from Trap

This instruction is defined by:

S

C.206.1. Encoding



C.206.2. Description

Returns from supervisor mode after handling a trap.

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

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

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

Table 15. Next privilege mode following an sret in (H)S-mode or M-mode

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

When the current privilege mode is VS-mode

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

Table 16. Next privilege mode following an sret in (H)S-mode or M-mode

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

C.206.3. Access

M	S	U
Always	Sometimes	Never

Access is determined as follows:

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

C.206.4. Decode Variables

C.206.5. IDL Operation

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

C.206.6. Sail Operation

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

C.206.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction
- VirtualInstruction

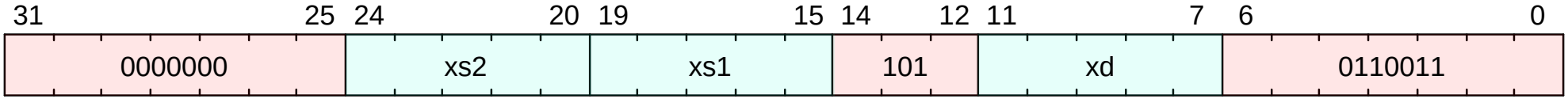
C.207. srl

Shift right logical

This instruction is defined by:

I

C.207.1. Encoding



C.207.2. Description

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

C.207.3. Access

M	S	U
Always	Always	Always

C.207.4. Decode Variables

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

C.207.5. IDL Operation

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

C.207.6. 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.207.7. Exceptions

This instruction does not generate synchronous exceptions.

C.208. srli

Shift right logical immediate

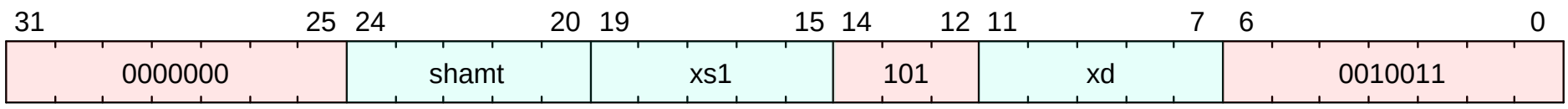
This instruction is defined by:

I

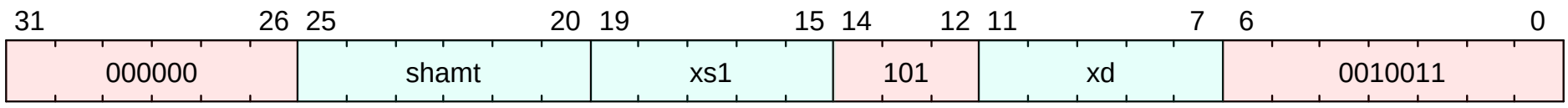
C.208.1. Encoding

This instruction has different encodings in RV32 and RV64.

RV32



RV64



C.208.2. Description

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

C.208.3. Access

M	S	U
Always	Always	Always

C.208.4. 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.208.5. IDL Operation

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

C.208.6. 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.208.7. Exceptions

This instruction does not generate synchronous exceptions.

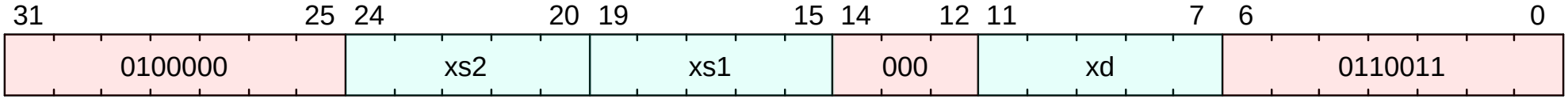
C.209. sub

Subtract

This instruction is defined by:

I

C.209.1. Encoding



C.209.2. Description

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

C.209.3. Access

M	S	U
Always	Always	Always

C.209.4. Decode Variables

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

C.209.5. IDL Operation

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

C.209.6. Sail Operation

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

C.209.7. Exceptions

This instruction does not generate synchronous exceptions.

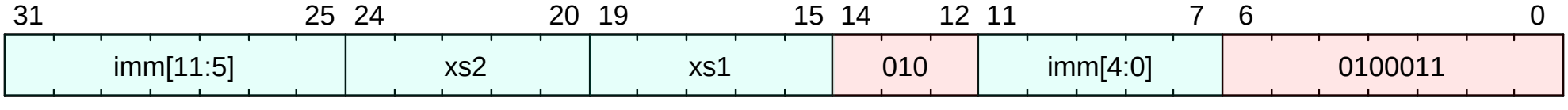
C.210. sw

Store word

This instruction is defined by:

I

C.210.1. Encoding



C.210.2. Description

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

C.210.3. Access

M	S	U
Always	Always	Always

C.210.4. Decode Variables

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

C.210.5. IDL Operation

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

C.210.6. 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.210.7. Exceptions

This instruction may result in the following synchronous exceptions:

- LoadAccessFault
- StoreAmoAccessFault
- StoreAmoAddressMisaligned
- StoreAmoPageFault

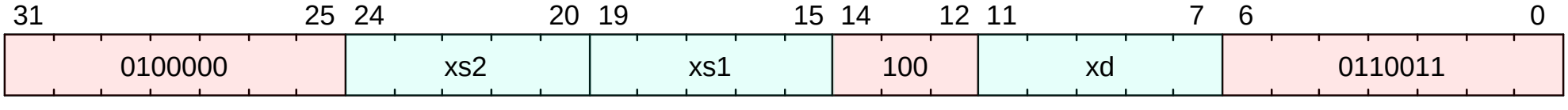
C.211. xnor

Exclusive NOR

This instruction is defined by:

(Zbb || Zbkb)

C.211.1. Encoding



C.211.2. Description

Performs the bit-wise exclusive-NOR operation on xs1 and xs2.

C.211.3. Access

M	S	U
Always	Always	Always

C.211.4. Decode Variables

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

C.211.5. IDL Operation

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

C.211.6. Sail Operation

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

C.211.7. Exceptions

This instruction may result in the following synchronous exceptions:

- IllegalInstruction

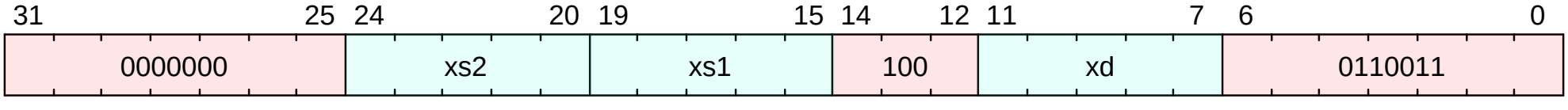
C.212. xor

Exclusive Or

This instruction is defined by:

I

C.212.1. Encoding



C.212.2. Description

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

C.212.3. Access

M	S	U
Always	Always	Always

C.212.4. Decode Variables

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

C.212.5. IDL Operation

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

C.212.6. Sail Operation

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

C.212.7. Exceptions

This instruction does not generate synchronous exceptions.

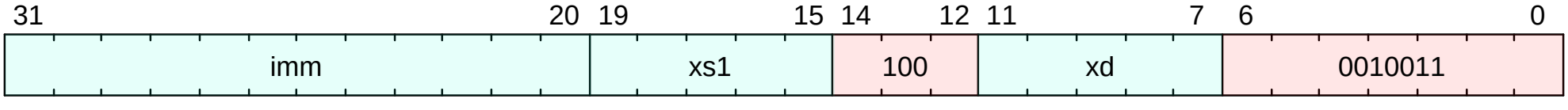
C.213. xori

Exclusive Or immediate

This instruction is defined by:

I

C.213.1. Encoding



C.213.2. Description

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

C.213.3. Access

M	S	U
Always	Always	Always

C.213.4. Decode Variables

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

C.213.5. IDL Operation

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

C.213.6. 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.213.7. 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
Length	64-bit
Privilege Mode	U

D.1.2. Format

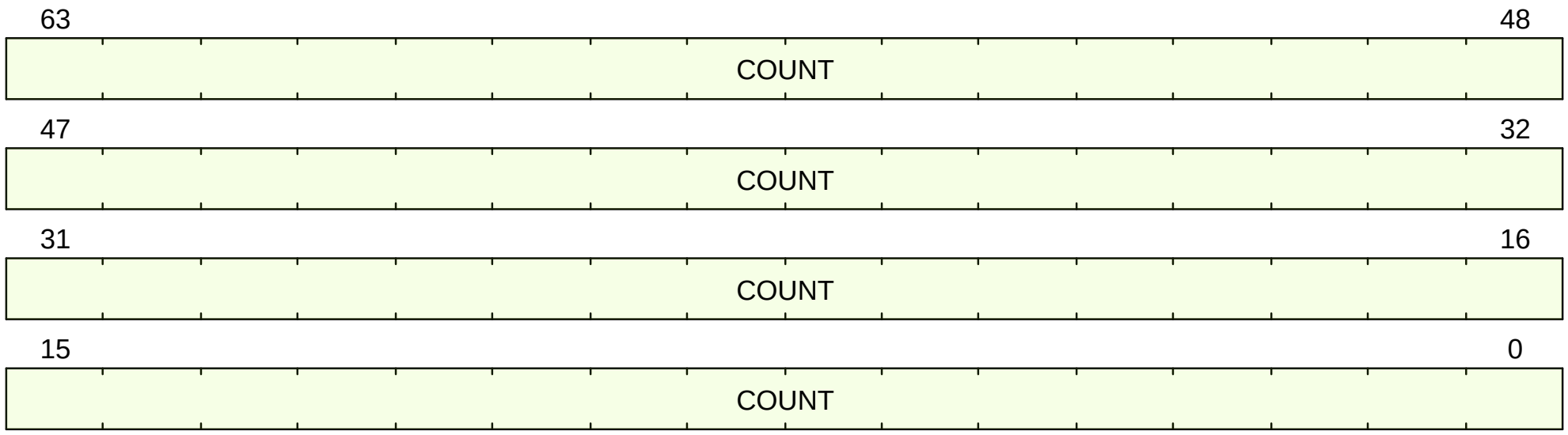


Figure 1. cycle format

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

D.2. 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 RVA22ProfileRelease.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 17. 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.2.1. Attributes

CSR Address	0x3
Defining extension	F
Length	32-bit
Privilege Mode	U

D.2.2. Format

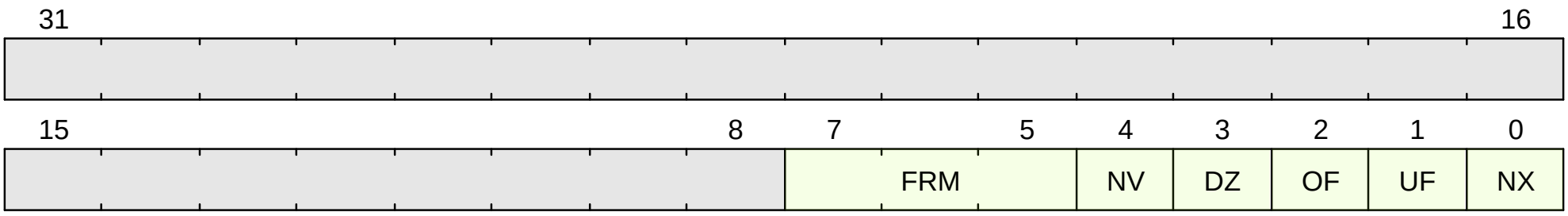


Figure 2. fcsr format

D.2.3. 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.2.4. 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.3. 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.3.1. Attributes

CSR Address	0x1
Defining extension	F
Length	32-bit
Privilege Mode	U

D.3.2. Format

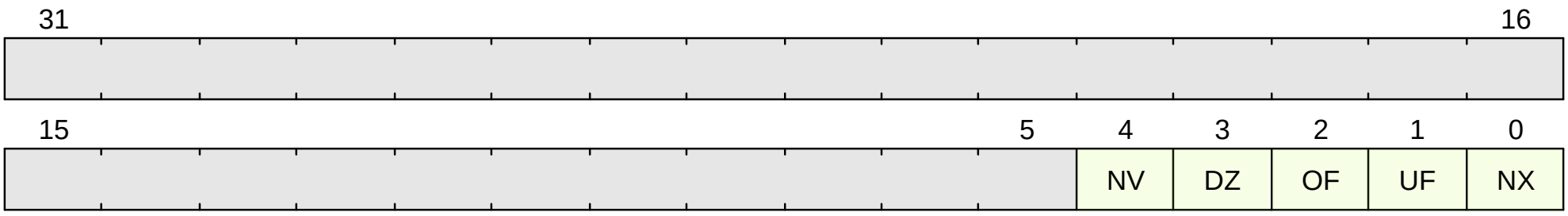


Figure 3. fflags format

D.3.3. 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.3.4. 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.3.5. 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.3.6. Software read

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

```
return (CSR[fcsr].NV `<< 4) | (CSR[fcsr].DZ `<< 3) | (CSR[fcsr].OF `<< 2) | (CSR[fcsr].UF `<< 1) | CSR[fcsr].NX;
```

D.4. frm

Floating-Point Dynamic Rounding Mode

Rounding modes are encoded as follows:

Table 18. 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.4.1. Attributes

CSR Address	0x2
Defining extension	F
Length	32-bit
Privilege Mode	U

D.4.2. Format

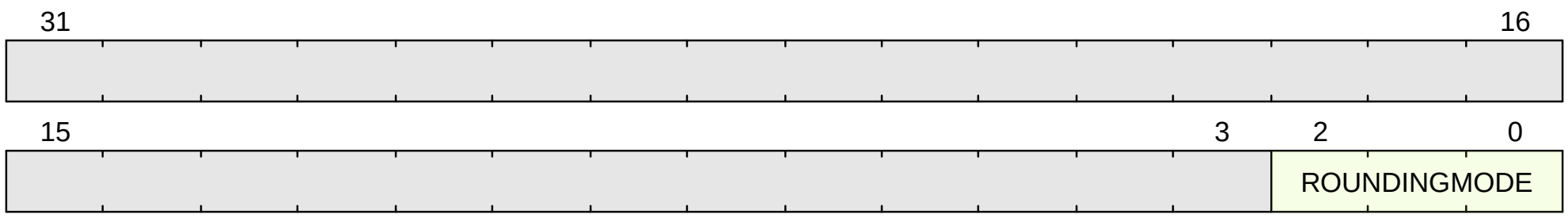


Figure 4. frm format

D.4.3. Field Summary

Name	Location	Type	Reset Value
frm.ROUNDINGMODE	2:0	RW-H	UNDEFINED_LEGAL

D.4.4. Fields

frm.ROUNDINGMODE Field

Location:
2:0

Description:
Rounding mode data.

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:

```
ROUNDINGMODE = CSR[fcsr].FRM = csr_value.ROUNDINGMODE;
return csr_value.ROUNDINGMODE;
```

D.4.6. Software read

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

```
return CSR[fcsr].FRM;
```

D.5. 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.5.1. Attributes

CSR Address	0xc0a
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.5.2. Format

Figure 5. hpmcounter10 format

D.5.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter10.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.5.4. Fields

hpmcounter10.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter10.COUNT.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

D.5.5. Software read

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

```
if (mode() == PrivilegeMode::S) {
  if (CSR[mcounteren].HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM10 & CSR[scounteren].HPM10) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM10 == 1'b0 && CSR[mcounteren].HPM10 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM10 & CSR[scounteren].HPM10) == 1'b0) && (CSR[mcounteren].HPM10 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM10 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(10);
```

D.6. 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.6.1. Attributes

CSR Address	0xc0b
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.6.2. Format

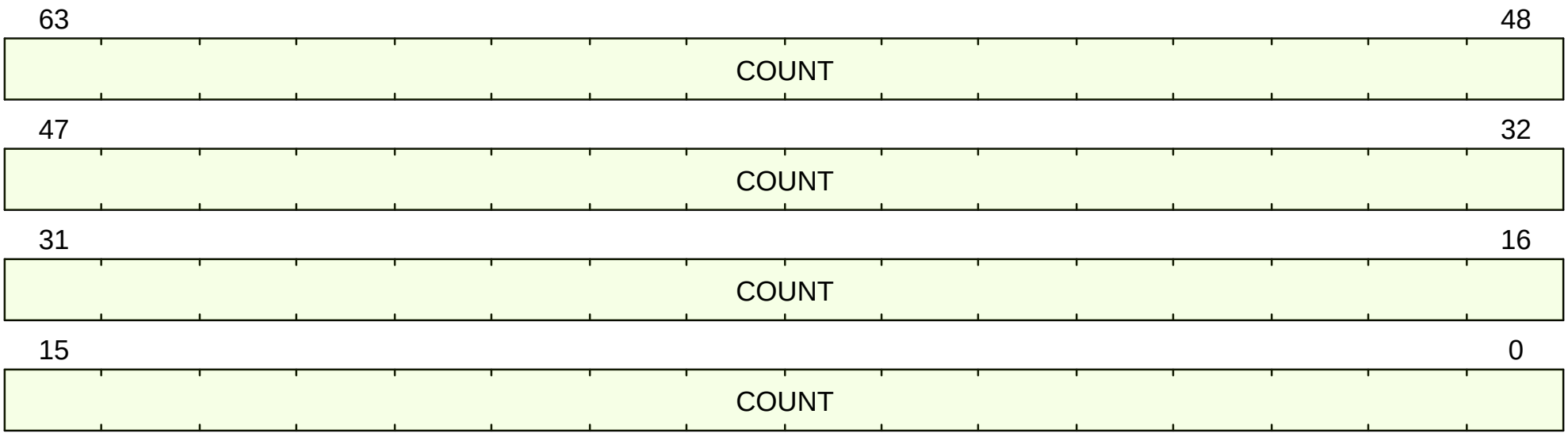


Figure 6. hpmcounter11 format

D.6.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter11.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.6.4. Fields

hpmcounter11.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter11.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 (CSR[mcounteren].HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM11 & CSR[scounteren].HPM11) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM11 == 1'b0 && CSR[mcounteren].HPM11 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM11 & CSR[scounteren].HPM11) == 1'b0) && (CSR[mcounteren].HPM11 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM11 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(11);
```

D.7. 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.7.1. Attributes

CSR Address	0xc0c
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.7.2. Format

Figure 7. hpmcounter12 format

D.7.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter12.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.7.4. Fields

hpmcounter12.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter12.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 (CSR[mcounteren].HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM12 & CSR[scounteren].HPM12) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM12 == 1'b0 && CSR[mcounteren].HPM12 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM12 & CSR[scounteren].HPM12) == 1'b0) && (CSR[mcounteren].HPM12 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM12 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(12);
```

D.8. 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.8.1. Attributes

CSR Address	0xc0d
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.8.2. Format

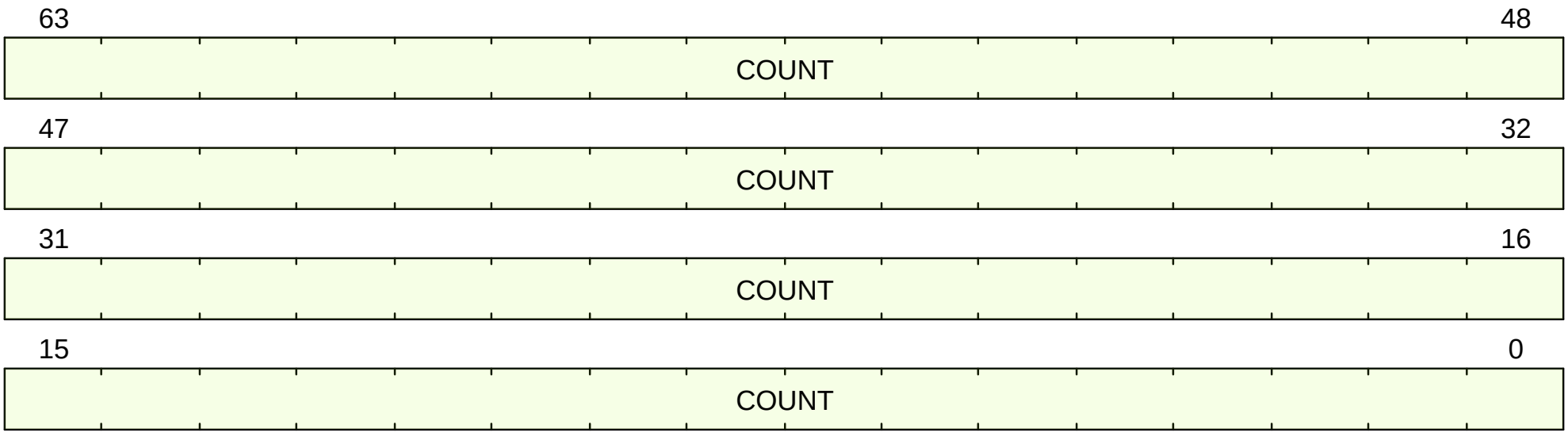


Figure 8. hpmcounter13 format

D.8.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter13.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.8.4. Fields

hpmcounter13.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter13.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 (CSR[mcounteren].HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM13 & CSR[scounteren].HPM13) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM13 == 1'b0 && CSR[mcounteren].HPM13 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM13 & CSR[scounteren].HPM13) == 1'b0) && (CSR[mcounteren].HPM13 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM13 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(13);
```

D.9. 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.9.1. Attributes

CSR Address	0xc0e
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.9.2. Format

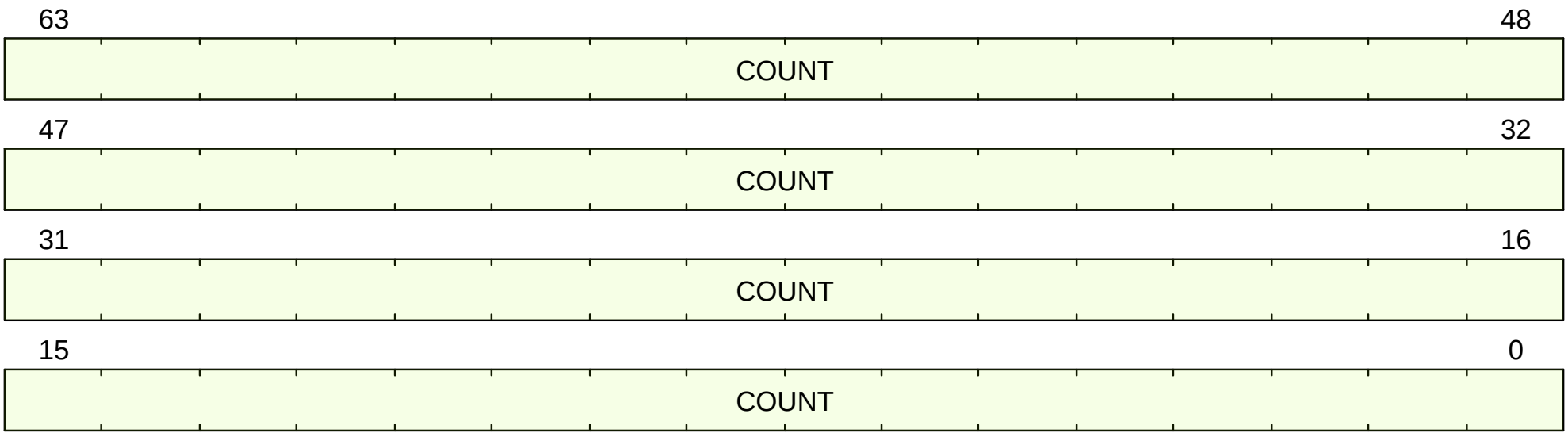


Figure 9. hpmcounter14 format

D.9.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter14.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.9.4. Fields

hpmcounter14.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter14.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 (CSR[mcounteren].HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM14 & CSR[scounteren].HPM14) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM14 == 1'b0 && CSR[mcounteren].HPM14 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM14 & CSR[scounteren].HPM14) == 1'b0) && (CSR[mcounteren].HPM14 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM14 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(14);
```

D.10. 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.10.1. Attributes

CSR Address	0xc0f
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.10.2. Format

Figure 10. hpmcounter15 format

D.10.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter15.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.10.4. Fields

hpmcounter15.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter15.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 (CSR[mcounteren].HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM15 & CSR[scounteren].HPM15) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM15 == 1'b0 && CSR[mcounteren].HPM15 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM15 & CSR[scounteren].HPM15) == 1'b0) && (CSR[mcounteren].HPM15 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM15 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(15);
```

D.11. 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.11.1. Attributes

CSR Address	0xc10
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.11.2. Format

Figure 11. hpmcounter16 format

D.11.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter16.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.11.4. Fields

hpmcounter16.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter16.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 (CSR[mcounteren].HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM16 & CSR[scounteren].HPM16) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM16 == 1'b0 && CSR[mcounteren].HPM16 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM16 & CSR[scounteren].HPM16) == 1'b0) && (CSR[mcounteren].HPM16 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM16 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(16);
```

D.12. 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.12.1. Attributes

CSR Address	0xc11
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.12.2. Format

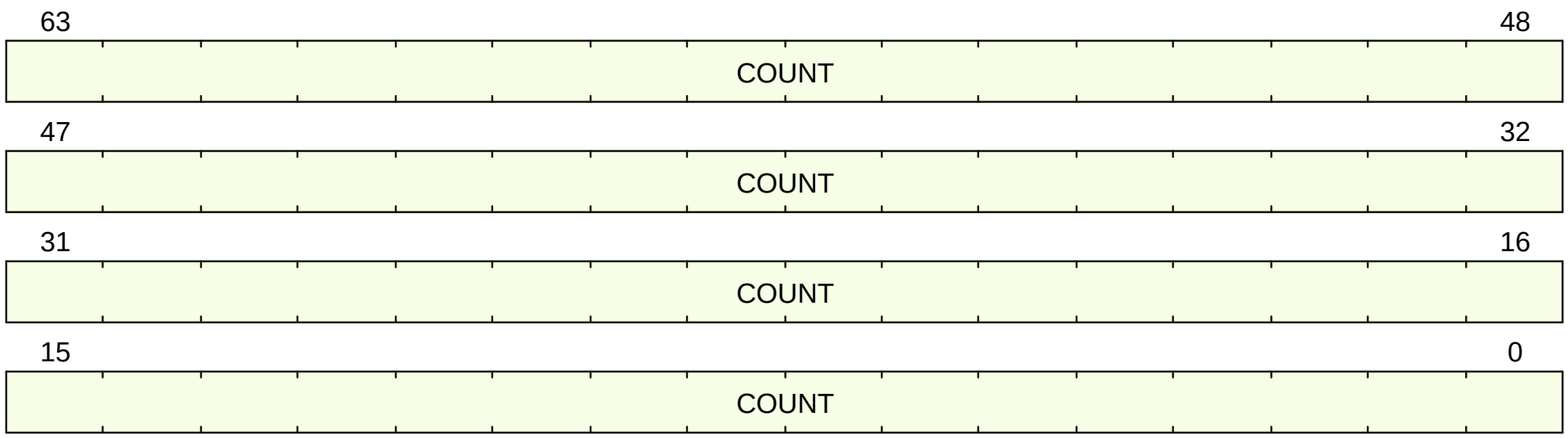


Figure 12. hpmcounter17 format

D.12.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter17.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.12.4. Fields

hpmcounter17.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter17.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 (CSR[mcounteren].HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM17 & CSR[scounteren].HPM17) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM17 == 1'b0 && CSR[mcounteren].HPM17 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM17 & CSR[scounteren].HPM17) == 1'b0) && (CSR[mcounteren].HPM17 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM17 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(17);
```

D.13. 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.13.1. Attributes

CSR Address	0xc12
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.13.2. Format

Figure 13. hpmcounter18 format

D.13.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter18.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.13.4. Fields

hpmcounter18.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter18.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 (CSR[mcounteren].HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM18 & CSR[scounteren].HPM18) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM18 == 1'b0 && CSR[mcounteren].HPM18 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM18 & CSR[scounteren].HPM18) == 1'b0) && (CSR[mcounteren].HPM18 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM18 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(18);
```

D.14. 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.14.1. Attributes

CSR Address	0xc13
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.14.2. Format

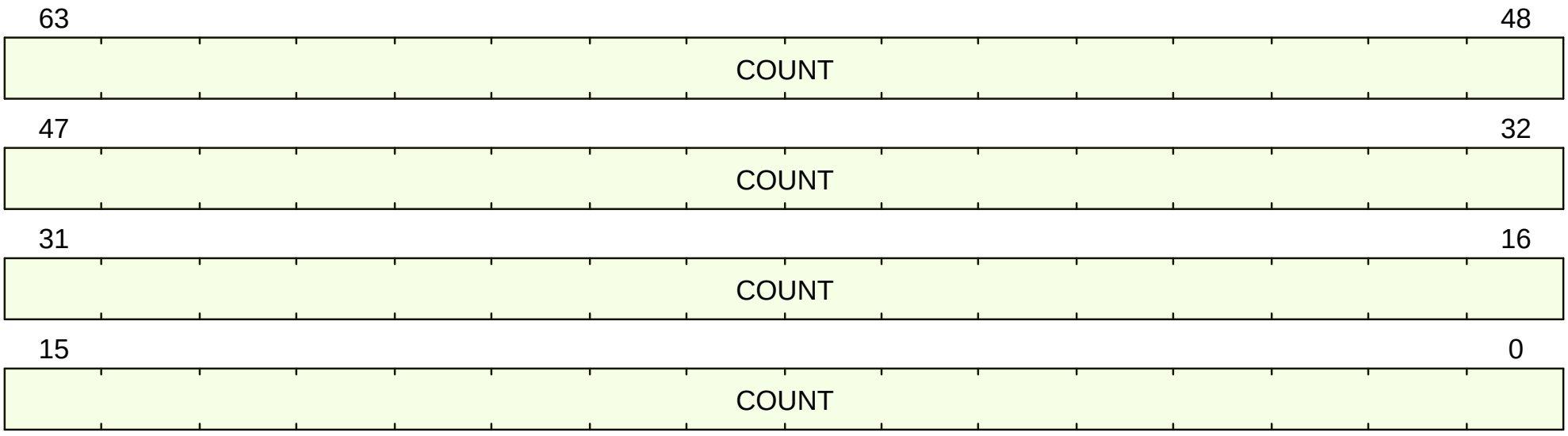


Figure 14. hpmcounter19 format

D.14.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter19.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.14.4. Fields

hpmcounter19.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter19.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 (CSR[mcounteren].HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM19 & CSR[scounteren].HPM19) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM19 == 1'b0 && CSR[mcounteren].HPM19 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM19 & CSR[scounteren].HPM19) == 1'b0) && (CSR[mcounteren].HPM19 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM19 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(19);
```

D.15. 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.15.1. Attributes

CSR Address	0xc14
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.15.2. Format

Figure 15. hpmcounter20 format

D.15.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter20.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.15.4. Fields

hpmcounter20.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter20.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 (CSR[mcounteren].HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM20 & CSR[scounteren].HPM20) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM20 == 1'b0 && CSR[mcounteren].HPM20 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM20 & CSR[scounteren].HPM20) == 1'b0) && (CSR[mcounteren].HPM20 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM20 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(20);
```

D.16. 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.16.1. Attributes

CSR Address	0xc15
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.16.2. Format

Figure 16. hpmcounter21 format

D.16.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter21.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.16.4. Fields

hpmcounter21.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter21.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 (CSR[mcounteren].HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM21 & CSR[scounteren].HPM21) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM21 == 1'b0 && CSR[mcounteren].HPM21 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM21 & CSR[scounteren].HPM21) == 1'b0) && (CSR[mcounteren].HPM21 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM21 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(21);
```

D.17. 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.17.1. Attributes

CSR Address	0xc16
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.17.2. Format

Figure 17. hpmcounter22 format

D.17.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter22.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.17.4. Fields

hpmcounter22.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter22.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 (CSR[mcounteren].HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM22 & CSR[scounteren].HPM22) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM22 == 1'b0 && CSR[mcounteren].HPM22 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM22 & CSR[scounteren].HPM22) == 1'b0) && (CSR[mcounteren].HPM22 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM22 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(22);
```

D.18. 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.18.1. Attributes

CSR Address	0xc17
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.18.2. Format

Figure 18. hpmcounter23 format

D.18.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter23.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.18.4. Fields

hpmcounter23.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter23.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 (CSR[mcounteren].HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM23 & CSR[scounteren].HPM23) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM23 == 1'b0 && CSR[mcounteren].HPM23 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM23 & CSR[scounteren].HPM23) == 1'b0) && (CSR[mcounteren].HPM23 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM23 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(23);
```

D.19. 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.19.1. Attributes

CSR Address	0xc18
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.19.2. Format

Figure 19. hpmcounter24 format

D.19.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter24.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.19.4. Fields

hpmcounter24.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter24.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 (CSR[mcounteren].HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM24 & CSR[scounteren].HPM24) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM24 == 1'b0 && CSR[mcounteren].HPM24 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM24 & CSR[scounteren].HPM24) == 1'b0) && (CSR[mcounteren].HPM24 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM24 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(24);
```

D.20. 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.20.1. Attributes

CSR Address	0xc19
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.20.2. Format

Figure 20. hpmcounter25 format

D.20.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter25.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.20.4. Fields

hpmcounter25.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter25.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 (CSR[mcounteren].HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM25 & CSR[scounteren].HPM25) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM25 == 1'b0 && CSR[mcounteren].HPM25 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM25 & CSR[scounteren].HPM25) == 1'b0) && (CSR[mcounteren].HPM25 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM25 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(25);
```

D.21. 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.21.1. Attributes

CSR Address	0xc1a
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.21.2. Format

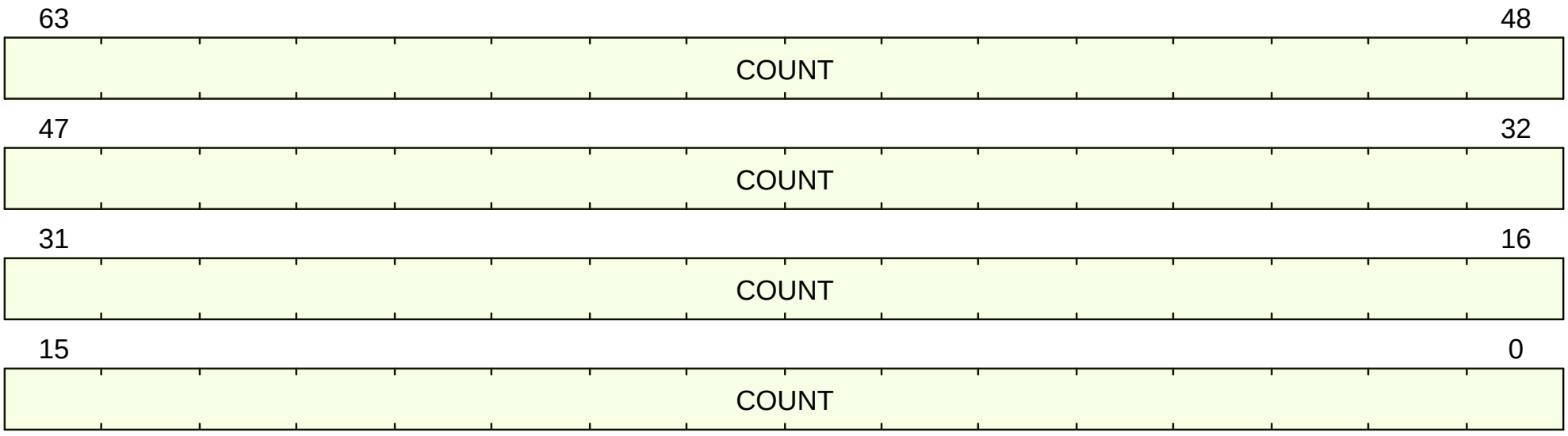


Figure 21. hpmcounter26 format

D.21.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter26.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.21.4. Fields

hpmcounter26.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter26.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 (CSR[mcounteren].HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM26 & CSR[scounteren].HPM26) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM26 == 1'b0 && CSR[mcounteren].HPM26 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM26 & CSR[scounteren].HPM26) == 1'b0) && (CSR[mcounteren].HPM26 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM26 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(26);
```

D.22. 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.22.1. Attributes

CSR Address	0xc1b
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.22.2. Format

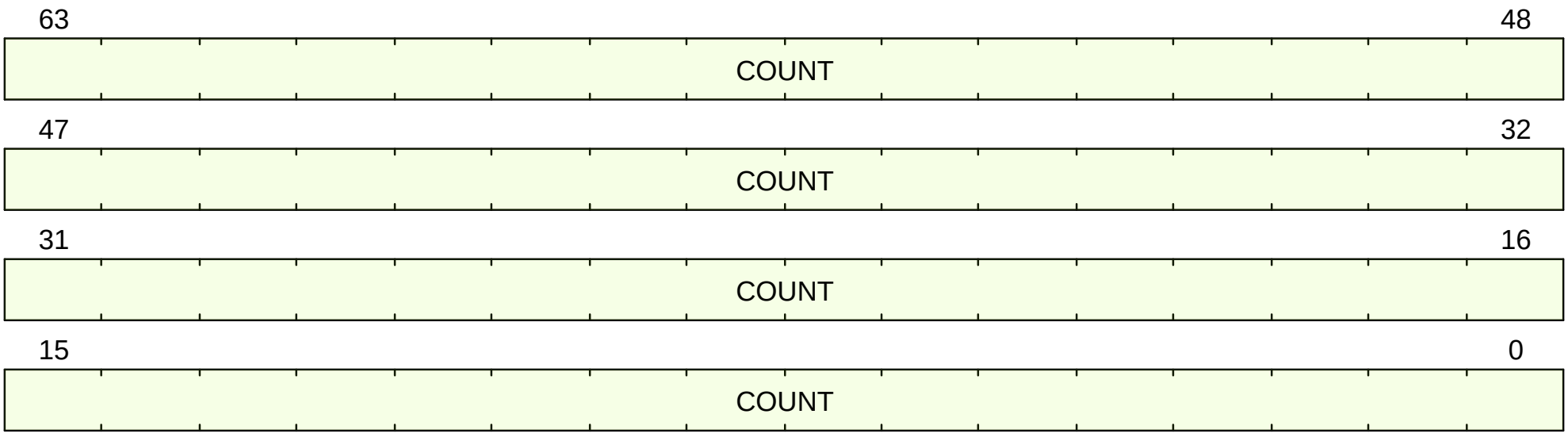


Figure 22. hpmcounter27 format

D.22.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter27.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.22.4. Fields

hpmcounter27.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter27.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 (CSR[mcounteren].HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM27 & CSR[scounteren].HPM27) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM27 == 1'b0 && CSR[mcounteren].HPM27 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM27 & CSR[scounteren].HPM27) == 1'b0) && (CSR[mcounteren].HPM27 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM27 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(27);
```

D.23. 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.23.1. Attributes

CSR Address	0xc1c
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.23.2. Format

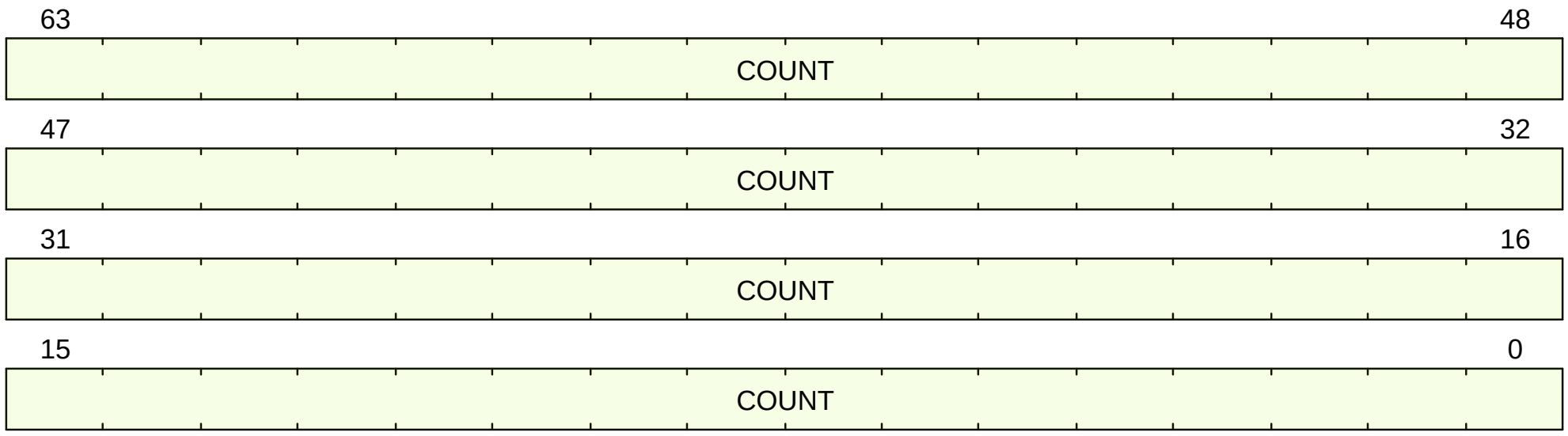


Figure 23. hpmcounter28 format

D.23.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter28.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.23.4. Fields

hpmcounter28.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter28.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 (CSR[mcounteren].HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM28 & CSR[scounteren].HPM28) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM28 == 1'b0 && CSR[mcounteren].HPM28 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM28 & CSR[scounteren].HPM28) == 1'b0) && (CSR[mcounteren].HPM28 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM28 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(28);
```

D.24. 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.24.1. Attributes

CSR Address	0xc1d
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.24.2. Format

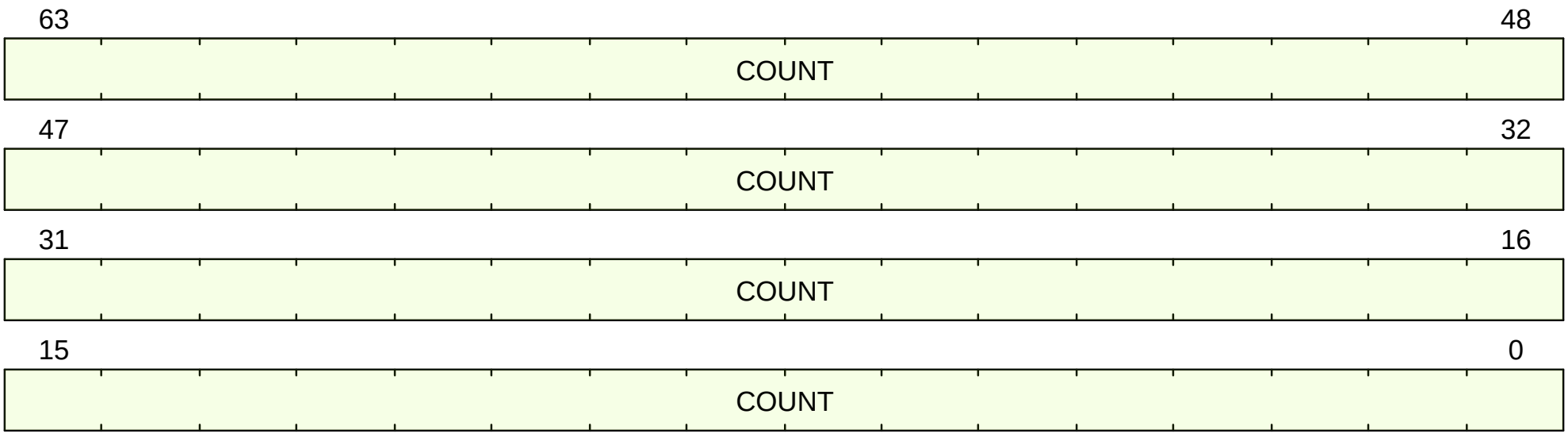


Figure 24. hpmcounter29 format

D.24.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter29.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.24.4. Fields

hpmcounter29.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter29.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 (CSR[mcounteren].HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM29 & CSR[scounteren].HPM29) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM29 == 1'b0 && CSR[mcounteren].HPM29 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM29 & CSR[scounteren].HPM29) == 1'b0) && (CSR[mcounteren].HPM29 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM29 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(29);
```

D.25. 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.25.1. Attributes

CSR Address	0xc03
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.25.2. Format

Figure 25. hpmcounter3 format

D.25.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter3.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.25.4. Fields

hpmcounter3.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter3.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 (CSR[mcounteren].HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM3 & CSR[scounteren].HPM3) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM3 == 1'b0 && CSR[mcounteren].HPM3 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM3 & CSR[scounteren].HPM3) == 1'b0) && (CSR[mcounteren].HPM3 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM3 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(3);
```

D.26. 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.26.1. Attributes

CSR Address	0xc1e
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.26.2. Format

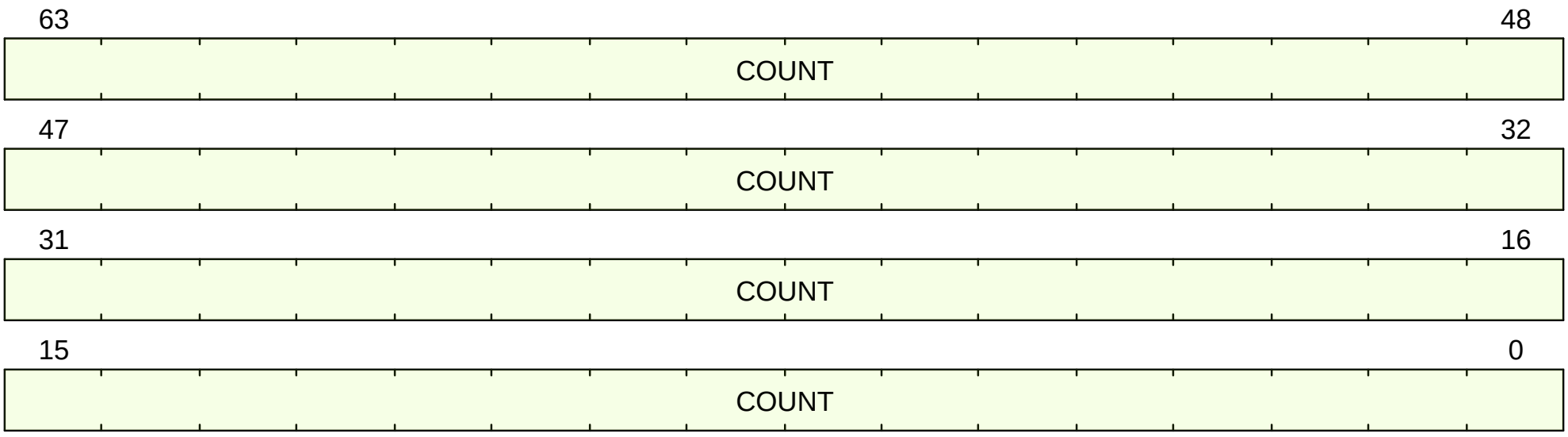


Figure 26. hpmcounter30 format

D.26.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter30.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.26.4. Fields

hpmcounter30.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter30.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 (CSR[mcounteren].HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM30 & CSR[scounteren].HPM30) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM30 == 1'b0 && CSR[mcounteren].HPM30 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM30 & CSR[scounteren].HPM30) == 1'b0) && (CSR[mcounteren].HPM30 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM30 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(30);
```

D.27. 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.27.1. Attributes

CSR Address	0xc1f
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.27.2. Format

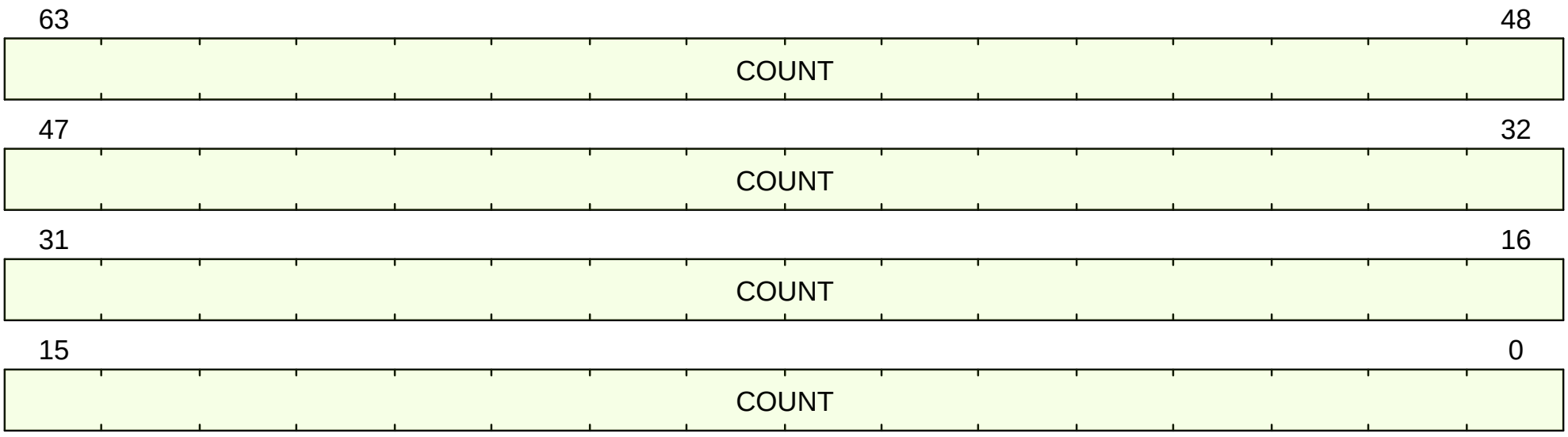


Figure 27. hpmcounter31 format

D.27.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter31.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.27.4. Fields

hpmcounter31.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter31.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 (CSR[mcounteren].HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM31 & CSR[scounteren].HPM31) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM31 == 1'b0 && CSR[mcounteren].HPM31 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM31 & CSR[scounteren].HPM31) == 1'b0) && (CSR[mcounteren].HPM31 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM31 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(31);
```

D.28. 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.28.1. Attributes

CSR Address	0xc04
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.28.2. Format

Figure 28. hpmcounter4 format

D.28.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter4.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.28.4. Fields

hpmcounter4.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter4.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 (CSR[mcounteren].HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM4 & CSR[scounteren].HPM4) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM4 == 1'b0 && CSR[mcounteren].HPM4 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM4 & CSR[scounteren].HPM4) == 1'b0) && (CSR[mcounteren].HPM4 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM4 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(4);
```

D.29. 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.29.1. Attributes

CSR Address	0xc05
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.29.2. Format

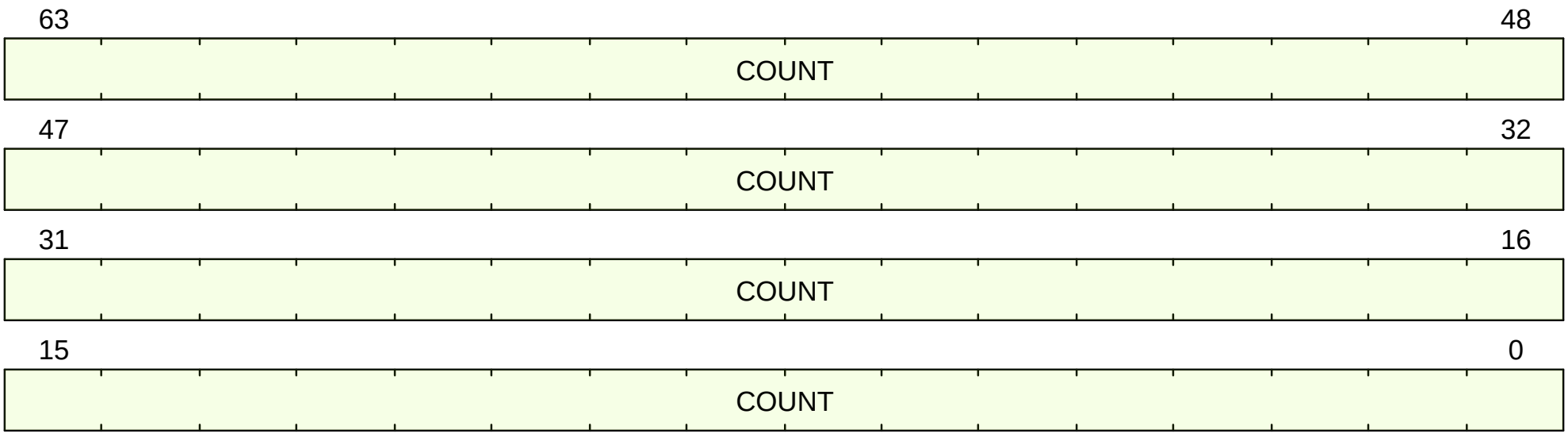


Figure 29. hpmcounter5 format

D.29.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter5.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.29.4. Fields

hpmcounter5.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter5.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 (CSR[mcounteren].HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM5 & CSR[scounteren].HPM5) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM5 == 1'b0 && CSR[mcounteren].HPM5 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM5 & CSR[scounteren].HPM5) == 1'b0) && (CSR[mcounteren].HPM5 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM5 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(5);
```

D.30. 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.30.1. Attributes

CSR Address	0xc06
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.30.2. Format

Figure 30. hpmcounter6 format

D.30.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter6.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.30.4. Fields

hpmcounter6.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter6.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 (CSR[mcounteren].HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM6 & CSR[scounteren].HPM6) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM6 == 1'b0 && CSR[mcounteren].HPM6 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM6 & CSR[scounteren].HPM6) == 1'b0) && (CSR[mcounteren].HPM6 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM6 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(6);
```

D.31. 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.31.1. Attributes

CSR Address	0xc07
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.31.2. Format

Figure 31. hpmcounter7 format

D.31.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter7.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.31.4. Fields

hpmcounter7.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter7.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 (CSR[mcounteren].HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM7 & CSR[scounteren].HPM7) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM7 == 1'b0 && CSR[mcounteren].HPM7 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM7 & CSR[scounteren].HPM7) == 1'b0) && (CSR[mcounteren].HPM7 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM7 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(7);
```

D.32. 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.32.1. Attributes

CSR Address	0xc08
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.32.2. Format

Figure 32. hpmcounter8 format

D.32.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter8.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.32.4. Fields

hpmcounter8.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter8.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 (CSR[mcounteren].HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM8 & CSR[scounteren].HPM8) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM8 == 1'b0 && CSR[mcounteren].HPM8 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM8 & CSR[scounteren].HPM8) == 1'b0) && (CSR[mcounteren].HPM8 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM8 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(8);
```

D.33. 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.33.1. Attributes

CSR Address	0xc09
Defining extension	Zihpm
Length	64-bit
Privilege Mode	U

D.33.2. Format

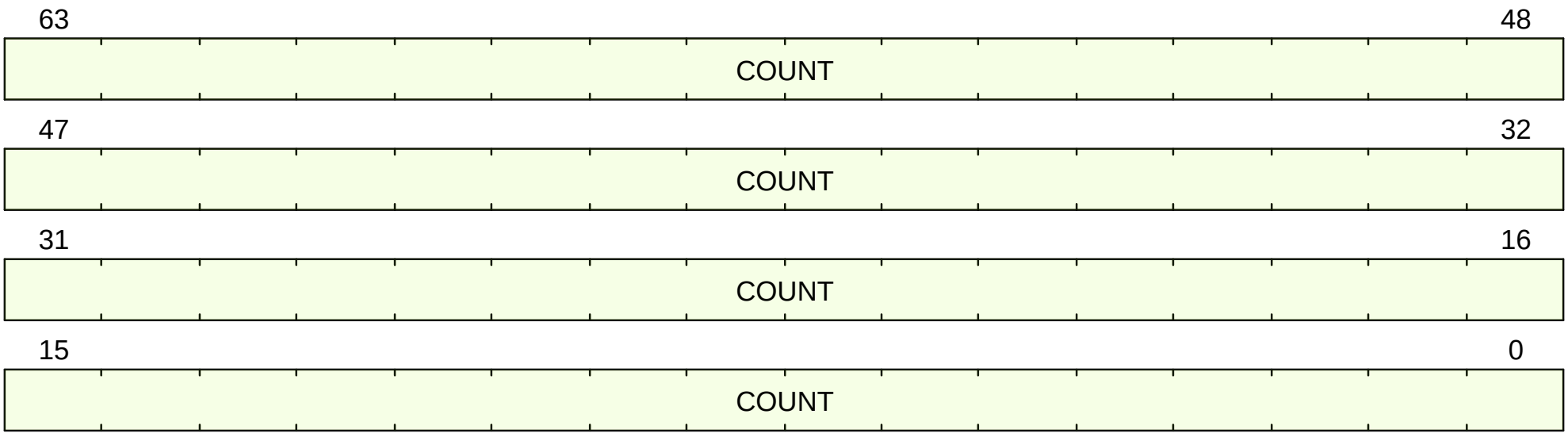


Figure 33. hpmcounter9 format

D.33.3. Field Summary

Name	Location	Type	Reset Value
hpmcounter9.COUNT	63:0	RO-H	UNDEFINED_LEGAL

D.33.4. Fields

hpmcounter9.COUNT Field

Location:

63:0

Description:

Alias of mhpmcounter9.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 (CSR[mcounteren].HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::U) {
  if (CSR[misa].S == 1'b1) {
    if ((CSR[mcounteren].HPM9 & CSR[scounteren].HPM9) == 1'b0) {
      raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
    }
  } else if (CSR[mcounteren].HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VS) {
  if (CSR[hcounteren].HPM9 == 1'b0 && CSR[mcounteren].HPM9 == 1'b1) {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
} else if (mode() == PrivilegeMode::VU) {
  if (CSR[hcounteren].HPM9 & CSR[scounteren].HPM9) == 1'b0) && (CSR[mcounteren].HPM9 == 1'b1 {
    raise(ExceptionCode::VirtualInstruction, mode(), $encoding);
  } else if (CSR[mcounteren].HPM9 == 1'b0) {
    raise(ExceptionCode::IllegalInstruction, mode(), $encoding);
  }
}
return read_hpm_counter(9);
```

D.34. 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.34.1. Attributes

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

D.34.2. Format

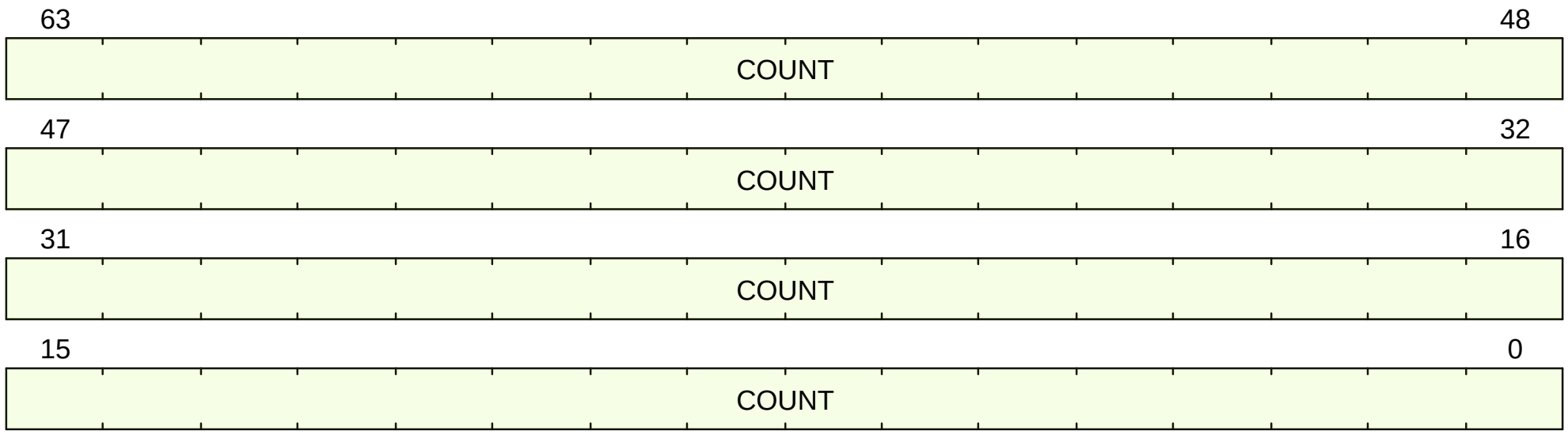


Figure 34. instret format

D.34.3. Field Summary

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

D.34.4. Fields

instret.COUNT Field

Location:

63:0

Description:

Alias of minstret.COUNT.

Type:

RO-H

Reset value:

0

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

D.35. mcounteren

Machine Counter Enable

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

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

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

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

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

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

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

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

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

D.35.1. Attributes

CSR Address	0x306
Defining extension	U
Length	32-bit
Privilege Mode	M

D.35.2. Format

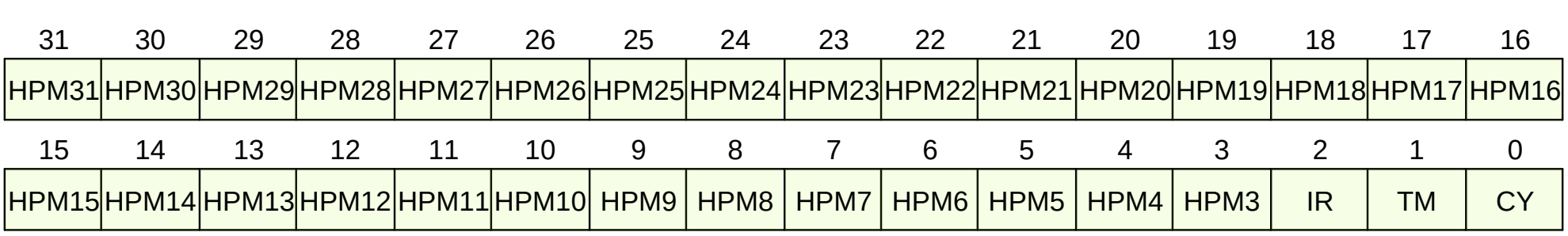


Figure 35. mcounteren format

D.35.3. Field Summary

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

Name	Location	Type	Reset Value
mcounteren.HPM3	3	RW RO	UNDEFINED_LEGAL
mcounteren.HPM4	4	RW RO	UNDEFINED_LEGAL
mcounteren.HPM5	5	RW RO	UNDEFINED_LEGAL
mcounteren.HPM6	6	RW RO	UNDEFINED_LEGAL
mcounteren.HPM7	7	RW RO	UNDEFINED_LEGAL
mcounteren.HPM8	8	RW RO	UNDEFINED_LEGAL
mcounteren.HPM9	9	RW RO	UNDEFINED_LEGAL
mcounteren.HPM10	10	RW RO	UNDEFINED_LEGAL
mcounteren.HPM11	11	RW RO	UNDEFINED_LEGAL
mcounteren.HPM12	12	RW RO	UNDEFINED_LEGAL
mcounteren.HPM13	13	RW RO	UNDEFINED_LEGAL
mcounteren.HPM14	14	RW RO	UNDEFINED_LEGAL
mcounteren.HPM15	15	RW RO	UNDEFINED_LEGAL
mcounteren.HPM16	16	RW RO	UNDEFINED_LEGAL
mcounteren.HPM17	17	RW RO	UNDEFINED_LEGAL
mcounteren.HPM18	18	RW RO	UNDEFINED_LEGAL
mcounteren.HPM19	19	RW RO	UNDEFINED_LEGAL
mcounteren.HPM20	20	RW RO	UNDEFINED_LEGAL
mcounteren.HPM21	21	RW RO	UNDEFINED_LEGAL
mcounteren.HPM22	22	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
mcounteren.HPM23	23	RW RO	UNDEFINED_LEGAL
mcounteren.HPM24	24	RW RO	UNDEFINED_LEGAL
mcounteren.HPM25	25	RW RO	UNDEFINED_LEGAL
mcounteren.HPM26	26	RW RO	UNDEFINED_LEGAL
mcounteren.HPM27	27	RW RO	UNDEFINED_LEGAL
mcounteren.HPM28	28	RW RO	UNDEFINED_LEGAL
mcounteren.HPM29	29	RW RO	UNDEFINED_LEGAL
mcounteren.HPM30	30	RW RO	UNDEFINED_LEGAL
mcounteren.HPM31	31	RW RO	UNDEFINED_LEGAL

D.35.4. Fields

mcounteren.CY Field

Location:

0

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.TM Field

Location:

1

Description:

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

Type:

RO

Reset value:

0

mcounteren.IR Field

Location:

2

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM3 Field

Location:

3

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM4 Field

Location:

4

Description:

When set, the hpmcounter4 CSR (an alias of mhpmcounter4) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM5 Field

Location:

5

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM6 Field

Location:

6

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM7 Field

Location:

7

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM8 Field

Location:

8

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM9 Field

Location:

9

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM10 Field

Location:

10

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM11 Field

Location:

11

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM12 Field

Location:

12

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM13 Field

Location:

13

Description:

When set, the hpmcounter13 CSR (an alias of mhpmcounter13) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM14 Field

Location:

14

Description:

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

<%- else -%>
U-mode.
<%- end -%>

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM15 Field

Location:

15

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM16 Field

Location:

16

Description:

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

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

When scounteren.HPM16 is also set, hpmcounter16 is further accessible to U-mode.
<%- end -%>

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM17 Field

Location:

17

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM18 Field

Location:

18

Description:

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

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

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

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

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM19 Field

Location:

19

Description:

When set, the hpmcounter19 CSR (an alias of mhpmcounter19) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM19 is also set, hpmcounter19 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM19 is also set, hpmcounter19 is further accessible to VS-mode.

When hcounteren.HPM19 && scounteren.HPM19 are both set, hpmcounter19 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM20 Field

Location:

20

Description:

When set, the hpmcounter20 CSR (an alias of mhpmcounter20) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM20 is also set, hpmcounter20 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM20 is also set, hpmcounter20 is further accessible to VS-mode.

When hcounteren.HPM20 && scounteren.HPM20 are both set, hpmcounter20 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM21 Field

Location:

21

Description:

When set, the hpmcounter21 CSR (an alias of mhpmcounter21) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM21 is also set, hpmcounter21 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM21 is also set, hpmcounter21 is further accessible to VS-mode.

When hcounteren.HPM21 && scounteren.HPM21 are both set, hpmcounter21 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM22 Field

Location:

22

Description:

When set, the hpmcounter22 CSR (an alias of mhpmcounter22) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM22 is also set, hpmcounter22 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM22 is also set, hpmcounter22 is further accessible to VS-mode.

When hcounteren.HPM22 && scounteren.HPM22 are both set, hpmcounter22 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM23 Field

Location:

23

Description:

When set, the hpmcounter23 CSR (an alias of mhpmcounter23) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM23 is also set, hpmcounter23 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM23 is also set, hpmcounter23 is further accessible to VS-mode.

When hcounteren.HPM23 && scounteren.HPM23 are both set, hpmcounter23 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM24 Field

Location:

24

Description:

When set, the hpmcounter24 CSR (an alias of mhpmcounter24) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM24 is also set, hpmcounter24 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM24 is also set, hpmcounter24 is further accessible to VS-mode.

When hcounteren.HPM24 && scounteren.HPM24 are both set, hpmcounter24 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM25 Field

Location:

25

Description:

When set, the hpmcounter25 CSR (an alias of mhpmcounter25) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM25 is also set, hpmcounter25 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM25 is also set, hpmcounter25 is further accessible to VS-mode.

When hcounteren.HPM25 && scounteren.HPM25 are both set, hpmcounter25 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM26 Field

Location:

26

Description:

When set, the hpmcounter26 CSR (an alias of mhpmcounter26) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM26 is also set, hpmcounter26 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM26 is also set, hpmcounter26 is further accessible to VS-mode.

When hcounteren.HPM26 && scounteren.HPM26 are both set, hpmcounter26 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM27 Field

Location:

27

Description:

When set, the hpmcounter27 CSR (an alias of mhpmcounter27) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM27 is also set, hpmcounter27 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM27 is also set, hpmcounter27 is further accessible to VS-mode.

When hcounteren.HPM27 && scounteren.HPM27 are both set, hpmcounter27 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM28 Field

Location:

28

Description:

When set, the hpmcounter28 CSR (an alias of mhpmcounter28) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM28 is also set, hpmcounter28 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM28 is also set, hpmcounter28 is further accessible to VS-mode.

When hcounteren.HPM28 && scounteren.HPM28 are both set, hpmcounter28 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM29 Field

Location:

29

Description:

When set, the hpmcounter29 CSR (an alias of mhpmcounter29) is accessible to
<%- if ext?(:S) -%>
S-mode.

<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM29 is also set, hpmcounter29 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM29 is also set, hpmcounter29 is further accessible to VS-mode.

When hcounteren.HPM29 && scounteren.HPM29 are both set, hpmcounter29 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM30 Field

Location:

30

Description:

When set, the hpmcounter30 CSR (an alias of mhpmcounter30) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

<%- if ext?(:S) -%>
When scounteren.HPM30 is also set, hpmcounter30 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM30 is also set, hpmcounter30 is further accessible to VS-mode.

When hcounteren.HPM30 && scounteren.HPM30 are both set, hpmcounter30 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

mcounteren.HPM31 Field

Location:

31

Description:

When set, the hpmcounter31 CSR (an alias of mhpmcounter31) is accessible to
<%- if ext?(:S) -%>
S-mode.
<%- else -%>
U-mode.
<%- end -%>

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

When scounteren.HPM31 is also set, hpmcounter31 is further accessible to U-mode.
<%- end -%>

<%- if ext?(:H) -%>
When hcounteren.HPM31 is also set, hpmcounter31 is further accessible to VS-mode.

When hcounteren.HPM31 && scounteren.HPM31 are both set, hpmcounter31 is further accessible to VU-mode.
<%- end -%>

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

D.36. medeleg

Machine Exception Delegation

Controls exception delegation from M-mode to (H)S-mode <%- if ext?(:H) -%> or, in conjunction with hedeleg, to VS-mode <%- end -%> .

An exception cause is delegated to (H)S-mode when all of the following hold:

- The corresponding field in medeleg is set.
- The current privilege level is not M-mode. <%- if ext?(:H) -%>
- The same field in hedeleg is clear. <%- end -%>

<%- if ext?(:H) -%> An exception cause is delegated to VS-mode when all of the following hold:

- The corresponding field in medeleg is set.
- The corresponding field in hedeleg is set.
- The current privilege level is not M-mode or HS-mode. <%- end -%>

Otherwise, an exception cause is handled by M-mode.

See interrupt documentation for more details.

D.36.1. Attributes

CSR Address	0x302
Defining extension	S
Length	64-bit
Privilege Mode	M

D.36.2. Format

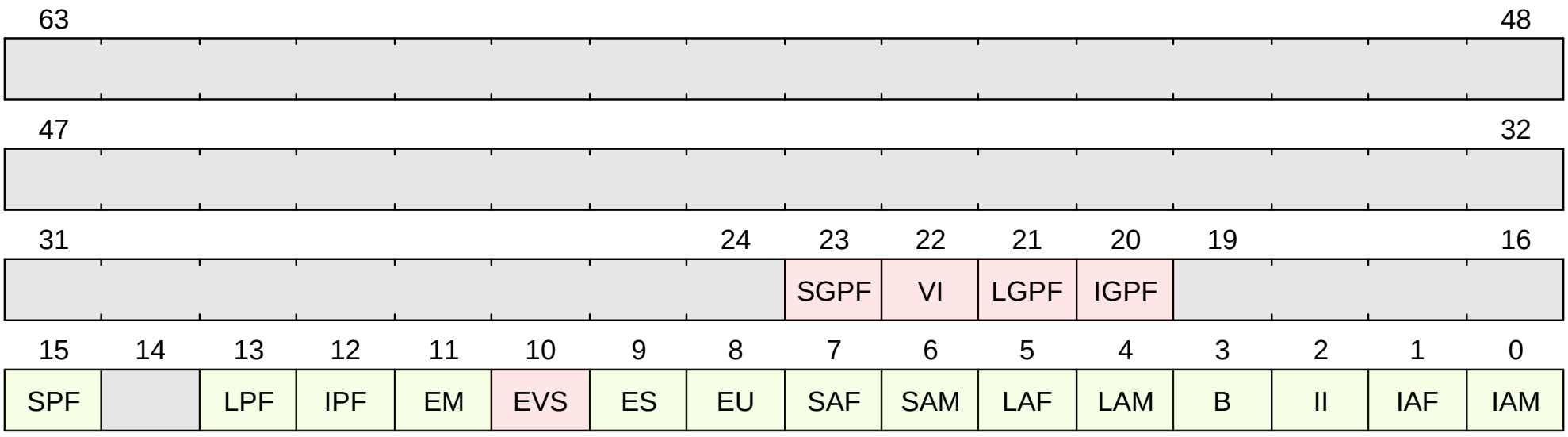


Figure 36. medeleg format

D.36.3. Field Summary

Name	Location	Type	Reset Value
medeleg.IAM	0	RW	UNDEFINED_LEGAL
medeleg.IAF	1	RW	UNDEFINED_LEGAL
medeleg.II	2	RW	UNDEFINED_LEGAL
medeleg.B	3	RW	UNDEFINED_LEGAL
medeleg.LAM	4	RW	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
medeleg.LAF	5	RW	UNDEFINED_LEGAL
medeleg.SAM	6	RW	UNDEFINED_LEGAL
medeleg.SAF	7	RW	UNDEFINED_LEGAL
medeleg.EU	8	RW	UNDEFINED_LEGAL
medeleg.ES	9	RW	UNDEFINED_LEGAL
medeleg.EVS	10	RW	UNDEFINED_LEGAL
medeleg.EM	11	RO	0
medeleg.IPF	12	RW	UNDEFINED_LEGAL
medeleg.LPF	13	RW	UNDEFINED_LEGAL
medeleg.SPF	15	RW	UNDEFINED_LEGAL
medeleg.IGPF	20	RW	UNDEFINED_LEGAL
medeleg.LGPF	21	RW	UNDEFINED_LEGAL
medeleg.VI	22	RW	UNDEFINED_LEGAL
medeleg.SGPF	23	RW	UNDEFINED_LEGAL

D.36.4. Fields

medeleg.IAM Field

Location:

0

Description:

Instruction Address Misaligned

Delegates Instruction Address Misaligned exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Address Misaligned exceptions may be further delegated to VS-mode if hedeleg.IAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

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

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAM .2+! hedeleg.IAM 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAM 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.IAF Field

Location:

1

Description:

Instruction Access Fault

Delegates Instruction Access Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Access Fault exceptions may be further delegated to VS-mode if hedeleg.IAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAF .2+! hedeleg.IAF 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IAF 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.II Field

Location:

2

Description:

Illegal Instruction

Delegates Illegal Instruction exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Illegal Instruction exceptions may be further delegated to VS-mode if hedeleg.II is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.II .2+! hedeleg.II 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.II 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.B Field

Location:

3

Description:

Breakpoint

Delegates Breakpoint exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Breakpoint exceptions may be further delegated to VS-mode if hedeleg.B is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]

!===
.2+! medeleg.B .2+! hedeleg.B 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.B 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LAM Field

Location:

4

Description:

Load Address Misaligned

Delegates Load Address Misaligned exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Address Misaligned exceptions may be further delegated to VS-mode if hedeleg.LAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

[when,"MISALIGNED_LDST == true"]
Note that because this implementation supports misaligned loads, this exception will never occur.
However, the writable bit should be presented anyway.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAM .2+! hedeleg.LAM 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAM 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LAF Field

Location:

5

Description:

Load Access Fault

Delegates Load Access Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Access Fault exceptions may be further delegated to VS-mode if hedeleg.LAF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAF .2+! hedeleg.LAF 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LAF 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SAM Field

Location:

6

Description:

Store/AMO Address Misaligned

Delegates Store/AMO Address Misaligned exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Address Misaligned exceptions may be further delegated to VS-mode if hedeleg.SAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

[when,"MISALIGNED_LDST == true && MISALIGNED_AMO == true"]

Note that because the implementation supports misaligned stores and misaligned AMOs (or no AMOs), this exception will never occur. Even so, the writable bit should be presented anyway.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAM .2+! hedeleg.SAM 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAM 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SAF Field

Location:

7

Description:

Store/AMO Access Fault

Delegates Store/AMO Access Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Access Fault exceptions may be further delegated to VS-mode if hedeleg.SAM is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAF .2+! hedeleg.SAF 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SAF 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M

! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.EU Field

Location:

8

Description:

Environment Call from U-Mode

Delegates Environment Call from U-mode exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Environment Call from U-mode exceptions may be further delegated to VS-mode if hedeleg.EU is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.EU .2+! hedeleg.EU 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.EU 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.ES Field

Location:

9

Description:

Environment Call from S-Mode

Delegates Environment Call from S-mode exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Environment Call from S-mode exceptions *cannot be delegated to VS-mode*.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.ES 2+^.>! Current Mode
.>h! `M-mode` .>h! `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! (H)S
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.EVS Field

Location:

10

Description:

Environment Call from VS-Mode

Delegates Environment Call from VS-mode exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Environment Call from S-mode exceptions *cannot be delegated to VS-mode*.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.EVS 2+^.>! Current Mode
.>h! `M-mode` .>h! `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! (H)S
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.EM Field

Location:

11

Description:

Environment Call from M-Mode

An Environment Call from M-mode cannot be delegated, so this is a read-only field.

All Environment Call from M-mode exceptions are taken by M-mode.

Type:

RO

Reset value:

0

medeleg.IPF Field

Location:

12

Description:

Instruction Page Fault

Delegates Instruction Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Page Fault exceptions may be further delegated to VS-mode if hedeleg.IPF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IPF .2+! hedeleg.IPF 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IPF 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LPF Field

Location:

13

Description:

Load Page Fault

Delegates Load Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Page Fault exceptions may be further delegated to VS-mode if hedeleg.LPF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]

!===
.2+! medeleg.LPF .2+! hedeleg.LPF 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LPF 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SPF Field

Location:

15

Description:

Store/AMO Page Fault

Delegates Store/AMO Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Page Fault exceptions may be further delegated to VS-mode if hedeleg.SPF is also set.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

<%- if ext?(:H) -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SPF .2+! hedeleg.SPF 4+^.>! Current Mode
.>h! M-mode .>h! HS-mode .>h! VS-mode .>h! VU-mode

! 0 ! 0 ! M ! M ! M ! M
! 0 ! 1 ! M ! M ! M ! M
! 1 ! 0 ! M ! HS ! HS ! HS
! 1 ! 1 ! M ! HS ! VS ! VS
!===
<%- else -%>
[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LPF 2+^.>! Current Mode
.>h! M-mode .>h! S-mode

! 0 ! M ! M
! 1 ! M ! S
!===
<%- end -%>

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.IGPF Field

Location:

20

Description:

Instruction Guest Page Fault

Delegates Instruction Guest Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Instruction Guest Page Fault exceptions *cannot be delegated to VS-mode*.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.IGPF 2+^.>! Current Mode
.>h! `M-mode` .>h! `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.LGPF Field

Location:

21

Description:

Load Guest Page Fault

Delegates Load Guest Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Load Guest Page Fault exceptions *cannot be delegated to VS-mode*.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.LGPF 2+^.>! Current Mode
.>h! `M-mode` .>h! `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.VI Field

Location:

22

Description:

Virtual Instruction

Delegates Virtual Instruction exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Virtual Instruction exceptions *cannot be delegated to VS-mode*.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.VI 2+^.>! Current Mode
.>h! `M-mode` .>h! `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

medeleg.SGPF Field

Location:

23

Description:

Store/AMO Guest Page Fault

Delegates Store/AMO Guest Page Fault exceptions to (H)S-mode.
<%- if ext?(:H) -%>
Store/AMO Guest Page Fault exceptions *cannot be delegated to VS-mode*.
<%- end -%>

Exceptions are never taken into a less-privileged mode, regardless of medeleg.

The handling mode is determined as follows:

[separator="!",%autowidth, %footer]
!===
.2+! medeleg.SGPF 2+^.>! Current Mode
.>h! `M-mode` .>h! `(H)S-mode`

! 0 ! M ! M
! 1 ! M ! HS
!===

Type:

RW

Reset value:

UNDEFINED_LEGAL

D.37. satp

Supervisor Address Translation and Protection

Controls the translation mode in (H)S-mode and U-mode, and holds the current ASID and page table base pointer.

D.37.1. Attributes

CSR Address	0x180
Defining extension	S
Length	* 32 when CSR[mstatus].SXL == 0 * 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.37.2. Format

This CSR format changes dynamically with XLEN.

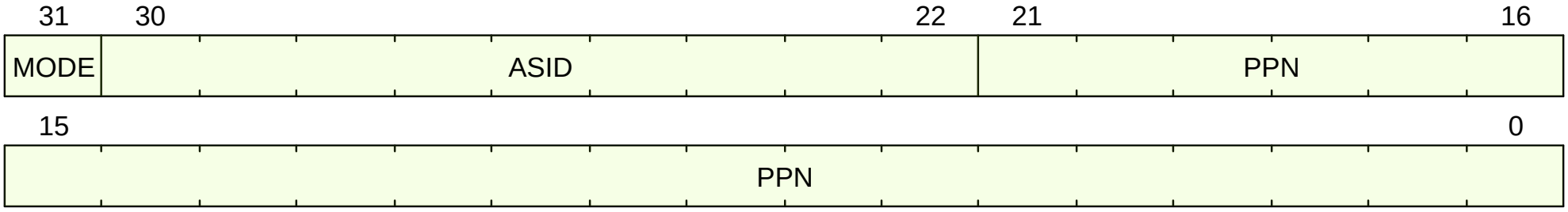


Figure 37. satp Format when CSR[mstatus].SXL == 0

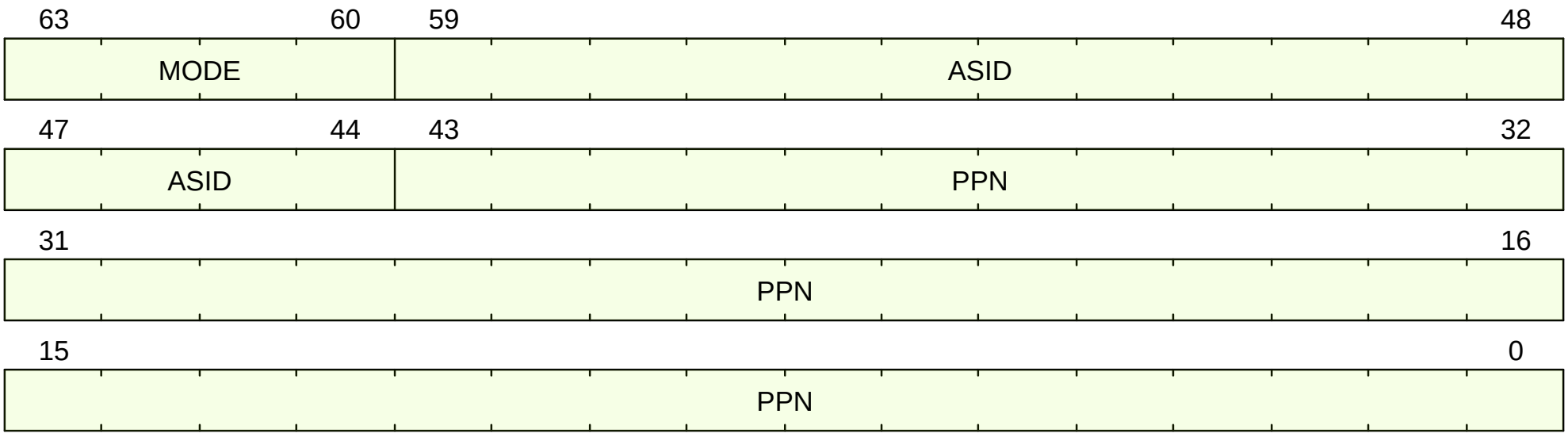


Figure 38. satp Format when CSR[mstatus].SXL == 1

D.37.3. Field Summary

Name	Location	Type	Reset Value
satp.MODE	31 63:60	RW-R	0
satp.ASID	30:22 59:44	RW-R	UNDEFINED_LEGAL
satp.PPN	21:0 43:0	RW-R	UNDEFINED_LEGAL

D.37.4. Fields

satp.MODE Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63:60 when CSR[mstatus].SXL == 1

Description:

Translation Mode

Controls the current translation mode according to the table below.

[separator="!",%autowidth]
!===
! Value ! Name ! Description

! 0 ! Bare a! No translation → virtual address == physical address
<%- if ext?(:Sv39) -%>
! 8 ! Sv39 ! 39-bit virtual address translation
<%- end -%>
<%- if ext?(:Sv48) -%>
! 9 ! Sv48 ! 48-bit virtual address translation
<%- end -%>
<%- if ext?(:Sv57) -%>
! 10 ! Sv57 ! 57-bit virtual address translation
<%- end -%>
!===

Any other value shall be ignored on a write.

Type:

RW-R

Reset value:

0

satp.ASID Field

Location:

- 30:22 when CSR[mstatus].SXL == 0
- 59:44 when CSR[mstatus].SXL == 1

Description:

Address Space ID

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

satp.PPN Field

Location:

- 21:0 when CSR[mstatus].SXL == 0
- 43:0 when CSR[mstatus].SXL == 1

Description:

Physical Page Number

The physical address of the active root page table is PPN << 12.

Can only hold values that correspond to a valid page table base, which will be implementation-dependent.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

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:

```
MODE = if (SATP_MODE_BARE && (csr_value.MODE == 0)) {
  # In Bare, ASID and PPN must be zero, else the entire write is ignored
  if (csr_value.ASID == 0 && csr_value.PPN == 0) {
    if (CSR[satp].MODE != 0) {
      # changes *to* Bare mode take effect immediately without needing sfence.vma
      # thus, an implicit sfence.vma occurs now
      VmaOrderType order_type;
      order_type.global = true;
      order_pgtbl_writes_before_vmafence(order_type);

      invalidate_translations(order_type);

      order_pgtbl_reads_after_vmafence(order_type);
    }
    return csr_value.MODE;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
}
else if (implemented?(ExtensionName::Sv32) && csr_value.MODE == 1) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }
  return csr_value.MODE;
}else if (implemented?(ExtensionName::Sv39) && csr_value.MODE == 8) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }
  return csr_value.MODE;
}
else if (implemented?(ExtensionName::Sv48) && csr_value.MODE == 9) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }

  return csr_value.MODE;
}
else if (implemented?(ExtensionName::Sv57) && csr_value.MODE == 10) {
  if (CSR[satp].MODE == 0) {
    # changes *from* Bare mode take effect immediately without needing sfence.vma
    # thus, an implicit sfence.vma occurs now
    VmaOrderType order_type;
    order_type.global = true;
    order_pgtbl_writes_before_vmafence(order_type);

    invalidate_translations(order_type);

    order_pgtbl_reads_after_vmafence(order_type);
  }
```

```
  return csr_value.MODE;
}
else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}

ASID = if (csr_value.MODE == 0) {
  # when MODE == Bare, PPN and ASID must be zero
  if (csr_value.ASID == 0 && csr_value.PPN == 0) {
    return csr_value.ASID;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else {
  XReg shamt = (xlen() == 32 || (CSR[mstatus].SXL == $bits(XRegWidth::XLEN32))) ? 9 : 16;
  XReg all_ones = ((MXLEN'1 << shamt) - 1);
  XReg largest_allowed_asid = (MXLEN'1 << shamt) - 1;

  if (csr_value.ASID == all_ones) {
    # the specification states that if all 1's are written to the ASID field, then
    # you must return the largest asid
    return largest_allowed_asid;
  } else if (csr_value.ASID > largest_allowed_asid) {
    # ... but is silent on what happens on any other illegal value
    return UNDEFINED_LEGAL_DETERMINISTIC;
  } else {
    # unrestricted
    return csr_value.ASID;
  }
}

PPN = if (csr_value.MODE == 0) {
  # when MODE == Bare, PPN and ASID must be zero
  if (csr_value.ASID == 0 && csr_value.PPN == 0) {
    return csr_value.PPN;
  } else {
    return UNDEFINED_LEGAL_DETERMINISTIC;
  }
} else {
  # unrestricted
  return csr_value.PPN;
}
```

D.38. scause

Supervisor Cause

Reports the cause of the latest exception.

D.38.1. Attributes

CSR Address	0x142
Defining extension	S
Length	* 32 when CSR[mstatus].SXL == 0 * 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.38.2. Format

This CSR format changes dynamically with XLEN.

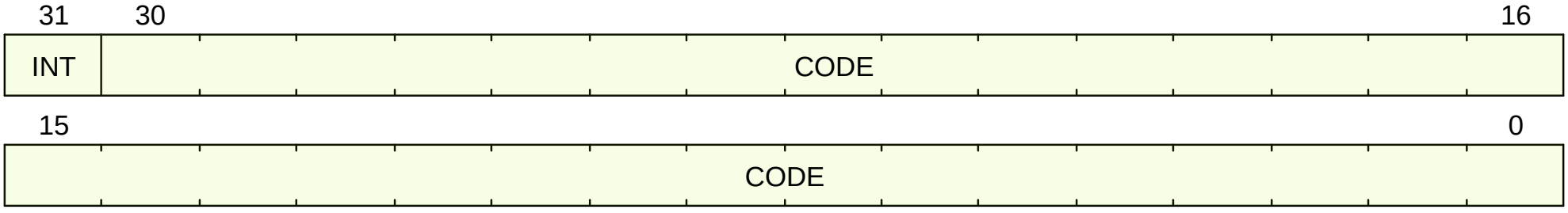


Figure 39. scause Format when CSR[mstatus].SXL == 0

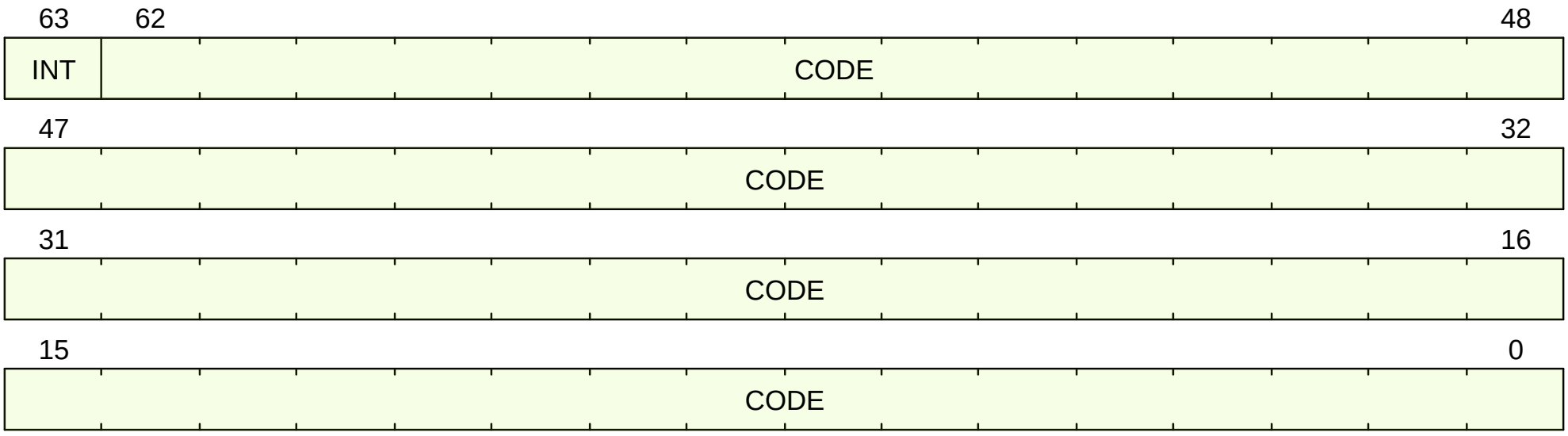


Figure 40. scause Format when CSR[mstatus].SXL == 1

D.38.3. Field Summary

Name	Location	Type	Reset Value
scause.INT	31 63	RW-RH	UNDEFINED_LEGAL
scause.CODE	30:0 62:0	RW-RH	UNDEFINED_LEGAL

D.38.4. Fields

scause.INT Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63 when CSR[mstatus].SXL == 1

Description:

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

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

scause.INT is writable.

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

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

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

scause.CODE Field

Location:

- 30:0 when CSR[mstatus].SXL == 0
- 62:0 when CSR[mstatus].SXL == 1

Description:

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

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

scause.CODE is writable.

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

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

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

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

Type:

RW-RH

Reset value:

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:

```
INT = # the write only holds if the INT/CODE combination is valid
# otherwise, the old value is retained
if (csr_value.INT == 1) {
  if (valid_interrupt_code?(csr_value.CODE)) {
    return 1;
  }
```

```
  return ILLEGAL_WLRL;
} else {
  if (valid_exception_code?(csr_value.CODE)) {
    return 1;
  }
  return ILLEGAL_WLRL;
}

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

D.39. scounteren

Supervisor Counter Enable

Delegates control of the hardware performance-monitoring counters to U-mode

D.39.1. Attributes

CSR Address	0x106
Defining extension	S
Length	32-bit
Privilege Mode	S

D.39.2. Format

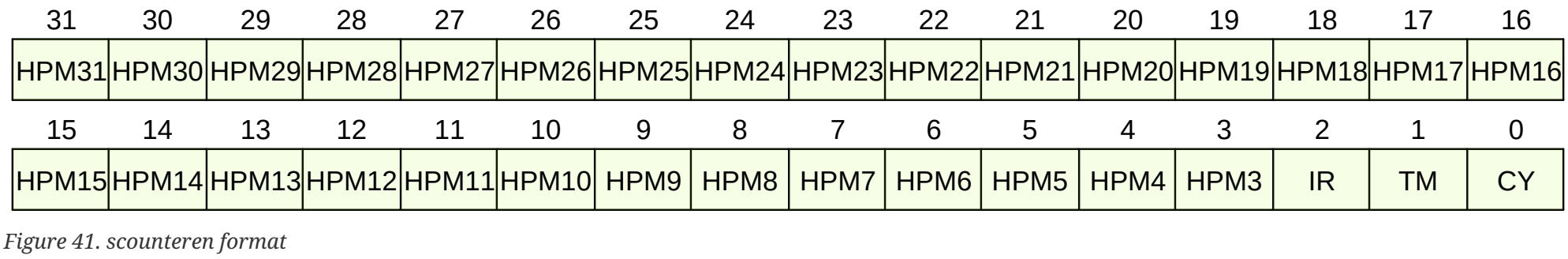


Figure 41. scounteren format

D.39.3. Field Summary

Name	Location	Type	Reset Value
scounteren.CY	0	RW RO	UNDEFINED_LEGAL
scounteren.TM	1	RW RO	UNDEFINED_LEGAL
scounteren.IR	2	RW RO	UNDEFINED_LEGAL
scounteren.HPM3	3	RW RO	UNDEFINED_LEGAL
scounteren.HPM4	4	RW RO	UNDEFINED_LEGAL
scounteren.HPM5	5	RW RO	UNDEFINED_LEGAL
scounteren.HPM6	6	RW RO	UNDEFINED_LEGAL
scounteren.HPM7	7	RW RO	UNDEFINED_LEGAL
scounteren.HPM8	8	RW RO	UNDEFINED_LEGAL
scounteren.HPM9	9	RW RO	UNDEFINED_LEGAL
scounteren.HPM10	10	RW RO	UNDEFINED_LEGAL
scounteren.HPM11	11	RW RO	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
scounteren.HPM12	12	RW RO	UNDEFINED_LEGAL
scounteren.HPM13	13	RW RO	UNDEFINED_LEGAL
scounteren.HPM14	14	RW RO	UNDEFINED_LEGAL
scounteren.HPM15	15	RW RO	UNDEFINED_LEGAL
scounteren.HPM16	16	RW RO	UNDEFINED_LEGAL
scounteren.HPM17	17	RW RO	UNDEFINED_LEGAL
scounteren.HPM18	18	RW RO	UNDEFINED_LEGAL
scounteren.HPM19	19	RW RO	UNDEFINED_LEGAL
scounteren.HPM20	20	RW RO	UNDEFINED_LEGAL
scounteren.HPM21	21	RW RO	UNDEFINED_LEGAL
scounteren.HPM22	22	RW RO	UNDEFINED_LEGAL
scounteren.HPM23	23	RW RO	UNDEFINED_LEGAL
scounteren.HPM24	24	RW RO	UNDEFINED_LEGAL
scounteren.HPM25	25	RW RO	UNDEFINED_LEGAL
scounteren.HPM26	26	RW RO	UNDEFINED_LEGAL
scounteren.HPM27	27	RW RO	UNDEFINED_LEGAL
scounteren.HPM28	28	RW RO	UNDEFINED_LEGAL
scounteren.HPM29	29	RW RO	UNDEFINED_LEGAL
scounteren.HPM30	30	RW RO	UNDEFINED_LEGAL
scounteren.HPM31	31	RW RO	UNDEFINED_LEGAL

D.39.4. Fields

scounteren.CY Field

Location:

0

Description:

When both scounteren.CY and mcounteren.CY are set, the cycle CSR (an alias of mcycle) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.CY)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.TM Field

Location:

1

Description:

When both scounteren.TM and mcounteren.TM are set, the time CSR (an alias of `mtime` memory-mapped CSR) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.TM)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.IR Field

Location:

2

Description:

When both scounteren.IR and mcounteren.IR are set, the instret CSR (an alias of memory-mapped minstret) is accessible to U-mode <% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.IR)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM3 Field

Location:

3

Description:

When both scounteren.HPM3 and mcounteren.HPM3 are set, the hpmcounter3 CSR (an alias of mhpmcounter3) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM3)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM4 Field

Location:

4

Description:

When both scounteren.HPM4 and mcounteren.HPM4 are set, the hpmcounter4 CSR (an alias of mhpmcounter4) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM4)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM5 Field

Location:

5

Description:

When both scounteren.HPM5 and mcounteren.HPM5 are set, the hpmcounter5 CSR (an alias of mhpmcounter5) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM5)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM6 Field

Location:

6

Description:

When both scounteren.HPM6 and mcounteren.HPM6 are set, the hpmcounter6 CSR (an alias of mhpmcounter6) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM6)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM7 Field

Location:

7

Description:

When both scounteren.HPM7 and mcounteren.HPM7 are set, the hpmcounter7 CSR (an alias of mhpmcounter7) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM7)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM8 Field

Location:

8

Description:

When both scounteren.HPM8 and mcounteren.HPM8 are set, the hpmcounter8 CSR (an alias of mhpmcounter8) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM8)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM9 Field

Location:

9

Description:

When both scounteren.HPM9 and mcounteren.HPM9 are set, the hpmcounter9 CSR (an alias of mhpmcounter9) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM9)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM10 Field

Location:

10

Description:

When both scounteren.HPM10 and mcounteren.HPM10 are set, the hpmcounter10 CSR (an alias of mhpmcounter10) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM10)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM11 Field

Location:

11

Description:

When both scounteren.HPM11 and mcounteren.HPM11 are set, the hpmcounter11 CSR (an alias of mhpmcounter11) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM11)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM12 Field

Location:

12

Description:

When both scounteren.HPM12 and mcounteren.HPM12 are set, the hpmcounter12 CSR (an alias of mhpmcounter12) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM12)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM13 Field

Location:

13

Description:

When both scounteren.HPM13 and mcounteren.HPM13 are set, the hpmcounter13 CSR (an alias of mhpmcounter13) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM13)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM14 Field

Location:

14

Description:

When both scounteren.HPM14 and mcounteren.HPM14 are set, the hpmcounter14 CSR (an alias of mhpmcounter14) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM14)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM15 Field

Location:

15

Description:

When both scounteren.HPM15 and mcounteren.HPM15 are set, the hpmcounter15 CSR (an alias of mhpmcounter15) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM15)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM16 Field

Location:

16

Description:

When both scounteren.HPM16 and mcounteren.HPM16 are set, the hpmcounter16 CSR (an alias of mhpmcounter16) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM16)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM17 Field

Location:

17

Description:

When both scounteren.HPM17 and mcounteren.HPM17 are set, the hpmcounter17 CSR (an alias of mhpmcounter17) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM17)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM18 Field

Location:

18

Description:

When both scounteren.HPM18 and mcounteren.HPM18 are set, the hpmcounter18 CSR (an alias of mhpmcounter18) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM18)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM19 Field

Location:

19

Description:

When both scounteren.HPM19 and mcounteren.HPM19 are set, the hpmcounter19 CSR (an alias of mhpmcounter19) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM19)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM20 Field

Location:

20

Description:

When both scounteren.HPM20 and mcounteren.HPM20 are set, the hpmcounter20 CSR (an alias of mhpmcounter20) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM20)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM21 Field

Location:

21

Description:

When both scounteren.HPM21 and mcounteren.HPM21 are set, the hpmcounter21 CSR (an alias of mhpmcounter21) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM21)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM22 Field

Location:

22

Description:

When both scounteren.HPM22 and mcounteren.HPM22 are set, the hpmcounter22 CSR (an alias of mhpmcounter22) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM22)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM23 Field

Location:

23

Description:

When both scounteren.HPM23 and mcounteren.HPM23 are set, the hpmcounter23 CSR (an alias of mhpmcounter23) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM23)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM24 Field

Location:

24

Description:

When both scounteren.HPM24 and mcounteren.HPM24 are set, the hpmcounter24 CSR (an alias of mhpmcounter24) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM24)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM25 Field

Location:

25

Description:

When both scounteren.HPM25 and mcounteren.HPM25 are set, the hpmcounter25 CSR (an alias of mhpmcounter25) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM25)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM26 Field

Location:

26

Description:

When both scounteren.HPM26 and mcounteren.HPM26 are set, the hpmcounter26 CSR (an alias of mhpmcounter26) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM26)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM27 Field

Location:

27

Description:

When both scounteren.HPM27 and mcounteren.HPM27 are set, the hpmcounter27 CSR (an alias of mhpmcounter27) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM27)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM28 Field

Location:

28

Description:

When both scounteren.HPM28 and mcounteren.HPM28 are set, the hpmcounter28 CSR (an alias of mhpmcounter28) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM28)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM29 Field

Location:

29

Description:

When both scounteren.HPM29 and mcounteren.HPM29 are set, the hpmcounter29 CSR (an alias of mhpmcounter29) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM29)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM30 Field

Location:

30

Description:

When both scounteren.HPM30 and mcounteren.HPM30 are set, the hpmcounter30 CSR (an alias of mhpmcounter30) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM30)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

scounteren.HPM31 Field

Location:

31

Description:

When both scounteren.HPM31 and mcounteren.HPM31 are set, the hpmcounter31 CSR (an alias of mhpmcounter31) is accessible to U-mode
<% if ext?(:H) %>(delegation to VS/VU mode is further handled by hcounteren.HPM31)<% end %>.

Type:

RW

RO

Reset value:

UNDEFINED_LEGAL

D.40. scountovf

Supervisor Count Overflow

A 32-bit read-only register that contains shadow copies of the OF bits in the 29 `mhpmevent` CSRs (mhpmevent3 - mhpmevent31) — where scountovf bit X corresponds to `mhpmeventX`.

This register enables supervisor-level overflow interrupt handler software to quickly and easily determine which counter(s) have overflowed without needing to make an execution environment call up to M-mode.

Read access to bit X is subject to the same mcounteren (or mcounteren and hcounteren) CSRs that mediate access to the `hpmcounter` CSRs by S-mode (or VS-mode).

In M-mode, scountovf bit X is always readable. In S/HS-mode, scountovf bit X is readable when mcounteren bit X is set, and otherwise reads as zero. Similarly, in VS-mode, it is readable when both mcounteren and hcounteren bit X are set.

D.40.1. Attributes

CSR Address	0xda0
Defining extension	Sscofpmf
Length	32-bit
Privilege Mode	S

D.40.2. Format

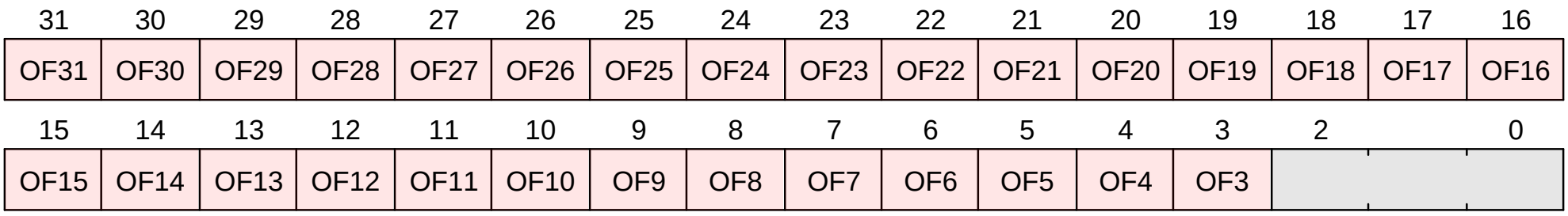


Figure 42. scountovf format

D.40.3. Field Summary

Name	Location	Type	Reset Value
scountovf.OF3	3	RO RO-H	UNDEFINED_LEGAL
scountovf.OF4	4	RO RO-H	UNDEFINED_LEGAL
scountovf.OF5	5	RO RO-H	UNDEFINED_LEGAL
scountovf.OF6	6	RO RO-H	UNDEFINED_LEGAL
scountovf.OF7	7	RO RO-H	UNDEFINED_LEGAL
scountovf.OF8	8	RO RO-H	UNDEFINED_LEGAL
scountovf.OF9	9	RO RO-H	UNDEFINED_LEGAL
scountovf.OF10	10	RO RO-H	UNDEFINED_LEGAL
scountovf.OF11	11	RO RO-H	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
scountovf.OF12	12	RO RO-H	UNDEFINED_LEGAL
scountovf.OF13	13	RO RO-H	UNDEFINED_LEGAL
scountovf.OF14	14	RO RO-H	UNDEFINED_LEGAL
scountovf.OF15	15	RO RO-H	UNDEFINED_LEGAL
scountovf.OF16	16	RO RO-H	UNDEFINED_LEGAL
scountovf.OF17	17	RO RO-H	UNDEFINED_LEGAL
scountovf.OF18	18	RO RO-H	UNDEFINED_LEGAL
scountovf.OF19	19	RO RO-H	UNDEFINED_LEGAL
scountovf.OF20	20	RO RO-H	UNDEFINED_LEGAL
scountovf.OF21	21	RO RO-H	UNDEFINED_LEGAL
scountovf.OF22	22	RO RO-H	UNDEFINED_LEGAL
scountovf.OF23	23	RO RO-H	UNDEFINED_LEGAL
scountovf.OF24	24	RO RO-H	UNDEFINED_LEGAL
scountovf.OF25	25	RO RO-H	UNDEFINED_LEGAL
scountovf.OF26	26	RO RO-H	UNDEFINED_LEGAL
scountovf.OF27	27	RO RO-H	UNDEFINED_LEGAL
scountovf.OF28	28	RO RO-H	UNDEFINED_LEGAL
scountovf.OF29	29	RO RO-H	UNDEFINED_LEGAL
scountovf.OF30	30	RO RO-H	UNDEFINED_LEGAL
scountovf.OF31	31	RO RO-H	UNDEFINED_LEGAL

D.40.4. Fields

scountovf.OF3 Field

Location:

3

Description:

[when="HPM_COUNTER_EN[3] == true"]
Shadow copy of mhpmevent3 overflow (OF) bit.

[when="HPM_COUNTER_EN[3] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF4 Field

Location:

4

Description:

[when="HPM_COUNTER_EN[4] == true"]
Shadow copy of mhpmevent4 overflow (OF) bit.

[when="HPM_COUNTER_EN[4] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF5 Field

Location:

5

Description:

[when="HPM_COUNTER_EN[5] == true"]
Shadow copy of mhpmevent5 overflow (OF) bit.

[when="HPM_COUNTER_EN[5] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF6 Field

Location:

6

Description:

[when="HPM_COUNTER_EN[6] == true"]
Shadow copy of mhpmevent6 overflow (OF) bit.

[when="HPM_COUNTER_EN[6] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF7 Field

Location:

7

Description:

[when="HPM_COUNTER_EN[7] == true"]
Shadow copy of mhpmevent7 overflow (OF) bit.

[when="HPM_COUNTER_EN[7] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF8 Field

Location:

8

Description:

[when="HPM_COUNTER_EN[8] == true"]
Shadow copy of mhpmevent8 overflow (OF) bit.

[when="HPM_COUNTER_EN[8] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF9 Field

Location:

9

Description:

[when="HPM_COUNTER_EN[9] == true"]
Shadow copy of mhpmevent9 overflow (OF) bit.

[when="HPM_COUNTER_EN[9] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF10 Field

Location:

10

Description:

[when="HPM_COUNTER_EN[10] == true"]
Shadow copy of mhpmevent10 overflow (OF) bit.

[when="HPM_COUNTER_EN[10] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF11 Field

Location:

11

Description:

[when="HPM_COUNTER_EN[11] == true"]
Shadow copy of mhpmevent11 overflow (OF) bit.

[when="HPM_COUNTER_EN[11] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF12 Field

Location:

12

Description:

[when="HPM_COUNTER_EN[12] == true"]
Shadow copy of mhpmevent12 overflow (OF) bit.

[when="HPM_COUNTER_EN[12] == false"]

This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF13 Field

Location:

13

Description:

[when="HPM_COUNTER_EN[13] == true"]
Shadow copy of mhpmevent13 overflow (OF) bit.

[when="HPM_COUNTER_EN[13] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF14 Field

Location:

14

Description:

[when="HPM_COUNTER_EN[14] == true"]
Shadow copy of mhpmevent14 overflow (OF) bit.

[when="HPM_COUNTER_EN[14] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF15 Field

Location:

15

Description:

[when="HPM_COUNTER_EN[15] == true"]
Shadow copy of mhpmevent15 overflow (OF) bit.

[when="HPM_COUNTER_EN[15] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF16 Field

Location:

16

Description:

[when="HPM_COUNTER_EN[16] == true"]
Shadow copy of mhpmevent16 overflow (OF) bit.

[when="HPM_COUNTER_EN[16] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF17 Field

Location:

17

Description:

[when="HPM_COUNTER_EN[17] == true"]
Shadow copy of mhpmevent17 overflow (OF) bit.

[when="HPM_COUNTER_EN[17] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF18 Field

Location:

18

Description:

[when="HPM_COUNTER_EN[18] == true"]
Shadow copy of mhpmevent18 overflow (OF) bit.

[when="HPM_COUNTER_EN[18] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF19 Field

Location:

19

Description:

[when="HPM_COUNTER_EN[19] == true"]
Shadow copy of mhpmevent19 overflow (OF) bit.

[when="HPM_COUNTER_EN[19] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF20 Field

Location:

20

Description:

[when="HPM_COUNTER_EN[20] == true"]
Shadow copy of mhpmevent20 overflow (OF) bit.

[when="HPM_COUNTER_EN[20] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF21 Field

Location:

21

Description:

[when="HPM_COUNTER_EN[21] == true"]
Shadow copy of mhpmevent21 overflow (OF) bit.

[when="HPM_COUNTER_EN[21] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF22 Field

Location:

22

Description:

[when="HPM_COUNTER_EN[22] == true"]
Shadow copy of mhpmevent22 overflow (OF) bit.

[when="HPM_COUNTER_EN[22] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF23 Field

Location:

23

Description:

[when="HPM_COUNTER_EN[23] == true"]
Shadow copy of mhpmevent23 overflow (OF) bit.

[when="HPM_COUNTER_EN[23] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF24 Field

Location:

24

Description:

[when="HPM_COUNTER_EN[24] == true"]
Shadow copy of mhpmevent24 overflow (OF) bit.

[when="HPM_COUNTER_EN[24] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF25 Field

Location:

25

Description:

[when="HPM_COUNTER_EN[25] == true"]
Shadow copy of mhpmevent25 overflow (OF) bit.

[when="HPM_COUNTER_EN[25] == false"]

This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF26 Field

Location:

26

Description:

[when="HPM_COUNTER_EN[26] == true"]
Shadow copy of mhpmevent26 overflow (OF) bit.

[when="HPM_COUNTER_EN[26] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF27 Field

Location:

27

Description:

[when="HPM_COUNTER_EN[27] == true"]
Shadow copy of mhpmevent27 overflow (OF) bit.

[when="HPM_COUNTER_EN[27] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF28 Field

Location:

28

Description:

[when="HPM_COUNTER_EN[28] == true"]
Shadow copy of mhpmevent28 overflow (OF) bit.

[when="HPM_COUNTER_EN[28] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF29 Field

Location:

29

Description:

[when="HPM_COUNTER_EN[29] == true"]
Shadow copy of mhpmevent29 overflow (OF) bit.

[when="HPM_COUNTER_EN[29] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF30 Field

Location:

30

Description:

[when="HPM_COUNTER_EN[30] == true"]
Shadow copy of mhpmevent30 overflow (OF) bit.

[when="HPM_COUNTER_EN[30] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

scountovf.OF31 Field

Location:

31

Description:

[when="HPM_COUNTER_EN[31] == true"]
Shadow copy of mhpmevent31 overflow (OF) bit.

[when="HPM_COUNTER_EN[31] == false"]
This field is read-only zero because the counter is not enabled.

Type:

RO

RO-H

Reset value:

UNDEFINED_LEGAL

D.40.5. Software read

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

```
Bits<32> mask;
if (mode() == PrivilegeMode::VS) {
  mask = $bits(CSR[CSR[mcounteren]]) & $bits(CSR[CSR[hcounteren]]);
} else {
  mask = $bits(CSR[CSR[mcounteren]]) & $bits(CSR[CSR[scounteren]]);
}
Bits<32> value = 0;
value = value | (CSR[mhpmevent3].OF << 3);
value = value | (CSR[mhpmevent4].OF << 4);
value = value | (CSR[mhpmevent5].OF << 5);
value = value | (CSR[mhpmevent6].OF << 6);
value = value | (CSR[mhpmevent7].OF << 7);
value = value | (CSR[mhpmevent8].OF << 8);
value = value | (CSR[mhpmevent9].OF << 9);
value = value | (CSR[mhpmevent10].OF << 10);
value = value | (CSR[mhpmevent11].OF << 11);
value = value | (CSR[mhpmevent12].OF << 12);
value = value | (CSR[mhpmevent13].OF << 13);
value = value | (CSR[mhpmevent14].OF << 14);
value = value | (CSR[mhpmevent15].OF << 15);
value = value | (CSR[mhpmevent16].OF << 16);
value = value | (CSR[mhpmevent17].OF << 17);
value = value | (CSR[mhpmevent18].OF << 18);
value = value | (CSR[mhpmevent19].OF << 19);
value = value | (CSR[mhpmevent20].OF << 20);
value = value | (CSR[mhpmevent21].OF << 21);
value = value | (CSR[mhpmevent22].OF << 22);
value = value | (CSR[mhpmevent23].OF << 23);
value = value | (CSR[mhpmevent24].OF << 24);
value = value | (CSR[mhpmevent25].OF << 25);
value = value | (CSR[mhpmevent26].OF << 26);
value = value | (CSR[mhpmevent27].OF << 27);
value = value | (CSR[mhpmevent28].OF << 28);
value = value | (CSR[mhpmevent29].OF << 29);
value = value | (CSR[mhpmevent30].OF << 30);
value = value | (CSR[mhpmevent31].OF << 31);
return value & mask;
```

D.41. sepc

Supervisor Exception Program Counter

Written with the PC of an instruction on an exception or interrupt taken in (H)S-mode.

Also controls where the hart jumps on an exception return from (H)S-mode.

D.41.1. Attributes

CSR Address	0x141
Defining extension	S
Length	64-bit
Privilege Mode	S

D.41.2. Format

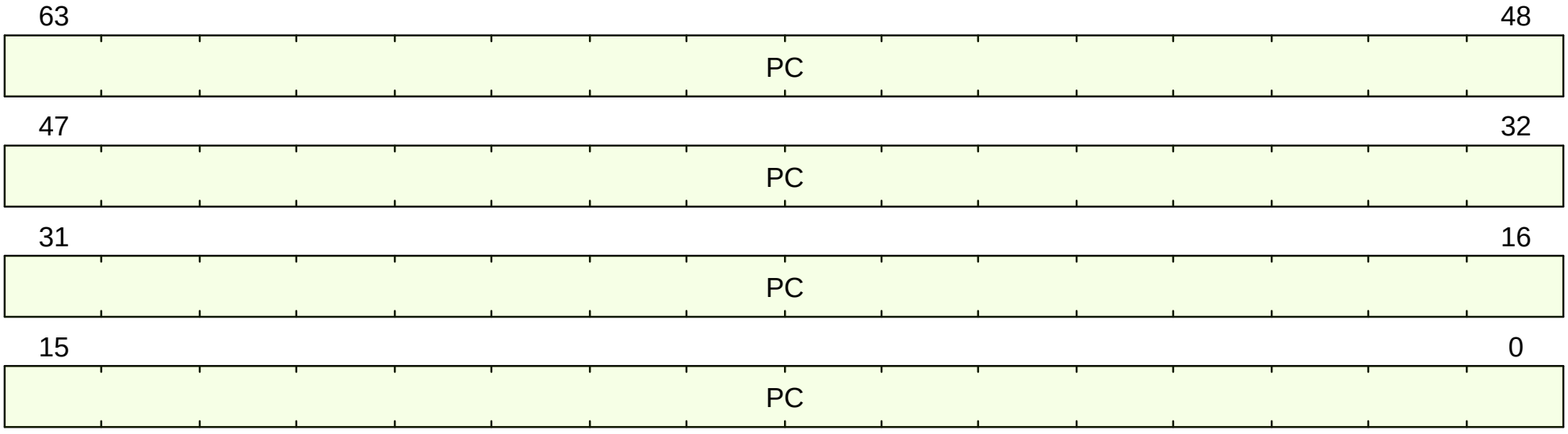


Figure 43. sepc format

D.41.3. Field Summary

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

D.41.4. Fields

sepc.PC Field

Location:

63:0

Description:

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

On an exception return from S-mode (from the SRET instruction),
control transfers to the virtual address read out of sepc.PC.

Because PCs are always <% if ext?(:C) %>halfword<% else %>word<% end %>-aligned,
<% if ext?(:C) %>bit 0<% else %>bits 1:0<% end %> of sepc.PC are always
read-only 0.

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

Holds bits 63:<%= ext?(:C) ? 2 : 1 %> of the virtual address associated with an exception.

Type:

RW-RH

Reset value:

UNDEFINED_LEGAL

D.41.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

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

D.41.6. Software read

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

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

D.42. sip

Supervisor Interrupt Pending

A restricted view of the interrupt pending bits in mip.

Hypervisor-related interrupts (VS-mode interrupts and Supervisor Guest interrupts) are not reflected in sip even though those interrupts can be taken in HS-mode. Instead, they are reported through `hip`.

D.42.1. Attributes

CSR Address	0x144
Defining extension	S
Length	64-bit
Privilege Mode	S

D.42.2. Format

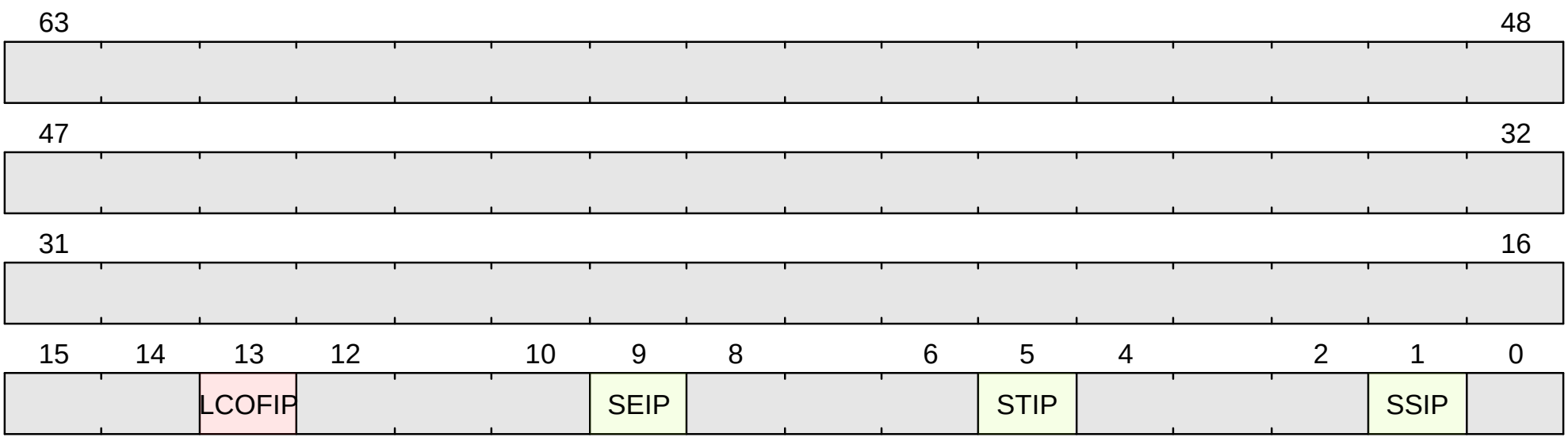


Figure 44. sip format

D.42.3. Field Summary

Name	Location	Type	Reset Value
sip.SSIP	1	RW	UNDEFINED_LEGAL
sip.STIP	5	RO-H	UNDEFINED_LEGAL
sip.SEIP	9	RO-H	UNDEFINED_LEGAL
sip.LCOFIP	13	RW-H	UNDEFINED_LEGAL

D.42.4. Fields

sip.SSIP Field

Location:

1

Description:

Supervisor Software Interrupt Pending

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

When Supervisor Software Interrupts are not delegated to (H)S-mode (mideleg.SSI is clear), sip.SSIP is read-only 0.

Otherwise, sip.SSIP is an alias of mip.SSIP.

<%- if ext?(:Smaia) -%>
When using AIA/IMSIC, IPIs are expected to be delivered as external interrupts

and SSIP is not backed by any hardware update (aside from any aliasing effects).

However, SSIP is still writable by S-mode software and, when written, can be used to
generate an S-mode Software Interrupt.
<%- end -%>

Since it is an alias, writes to sip.SSIP are also be reflected in mip.SSIP<% if ext?(:Smaia) %> and `mvip.SSIP`<% end %>.

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

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

  To summarize:
  [separator="!",%autowidth]
  !===
  ! mideleg.SSI ! sip.SSIP behavior

  ! 0 ! read-only 0
  ! 1 ! writable alias of mip.SSIP <% if ext?(:Smaia) %>and `mvip.SSIP`<% end %>
  !===

Type:

RW

Reset value:

UNDEFINED_LEGAL

sip.STIP Field

Location:

5

Description:

Supervisor Timer Interrupt Pending

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

When Supervisor Timer Interrupts are not delegated to (H)S-mode (*i.e.*, mideleg.STI is clear), sip.STIP is read-only 0.

Otherwise, sip.STIP is a read-only view of mip.STIP.

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

- mip.STIP when mideleg.STI is set
  <%- if ext?(:Smaia) -%>
- `mvip.STIP` when mideleg.SSI is set and menvcfg.STCE is clear.
  <%- end -%>

  To summarize:
  [separator="!",%autowidth]
  !===
  ! mideleg.STI ! sip.STIP behavior

  ! 0 ! read-only 0
  ! 1 ! read-only alias of mip.STIP <% if ext?(:Smaia) %>(and `mvip.STIP` when menvcfg.STCE is clear)<% end %>
  !===

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sip.SEIP Field

Location:

9

Description:

Supervisor External Interrupt Pending

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

When Supervisor External Interrupts are not delegated to (H)S-mode (*i.e.*, mideleg.SEI is clear), sip.SEIP is read-only 0.

Otherwise, sip.SEIP is a read-only view of mip.SEIP.

To summarize:
[separator="!",%autowidth]
!===
! mideleg.SEI ! sip.SEIP behavior

! 0 ! read-only 0
! 1 ! read-only alias of mip.SEIP
!===

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sip.LCOFIP Field

Location:

13

Description:

Local Counter Overflow Interrupt pending

Reports the current pending state of a Local Counter Overflow interrupt.

When Local Counter Overflow interrupts are not delegated to (H)S-mode (*i.e.*, mideleg.LCOFI is clear), sip.LCOFIP is read-only 0.

Otherwise, sip.LCOFIP is an alias of mip.LCOFIP.

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

To summarize:
[separator="!",%autowidth]
!===
! mideleg.LCOFI ! sip.LCOFIP behavior

! 0 ! read-only 0
! 1
a! writable alias of mip.LCOFIP (and `vsip.LCOFIP` when `hideleg.LCOFI` is set)
!===

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

D.43. sscratch

Supervisor Scratch Register

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

D.43.1. Attributes

CSR Address	0x140
Defining extension	S
Length	64-bit
Privilege Mode	S

D.43.2. Format

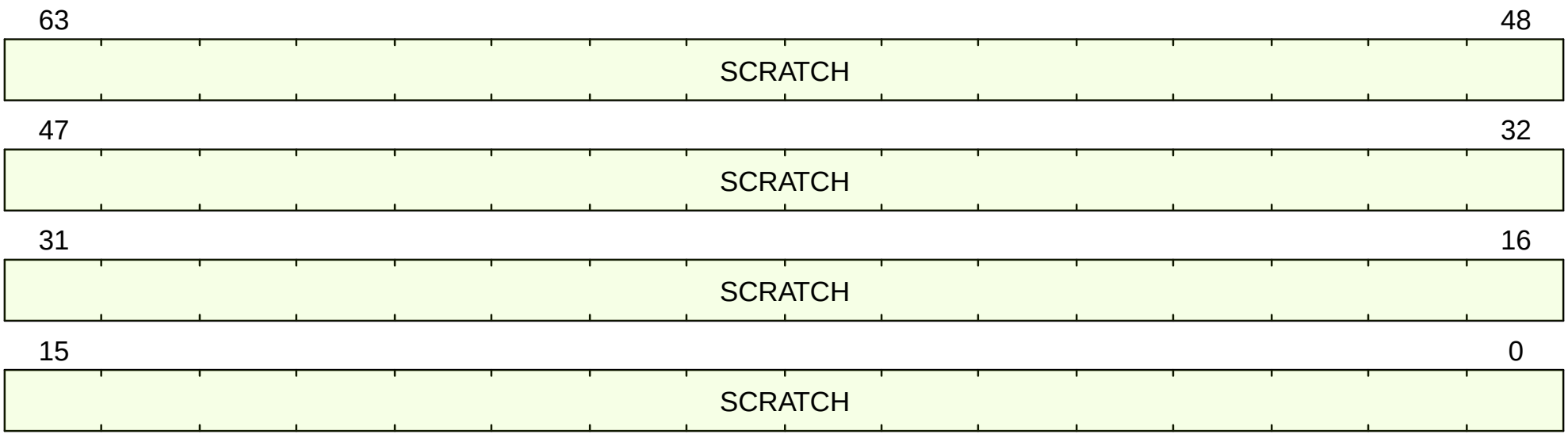


Figure 45. sscratch format

D.43.3. Field Summary

Name	Location	Type	Reset Value
sscratch.SCRATCH	63:0	RW	UNDEFINED_LEGAL

D.43.4. Fields

sscratch.SCRATCH Field

Location:

63:0

Description:

Scratch value

Type:

RW

Reset value:

UNDEFINED_LEGAL

D.44. sstatus

Supervisor Status

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

All fields in sstatus are aliases of the same field in mstatus.

D.44.1. Attributes

CSR Address	0x100
Defining extension	S
Length	* 32 when CSR[mstatus].SXL == 0 * 64 when CSR[mstatus].SXL == 1
Privilege Mode	S

D.44.2. Format

This CSR format changes dynamically with XLEN.

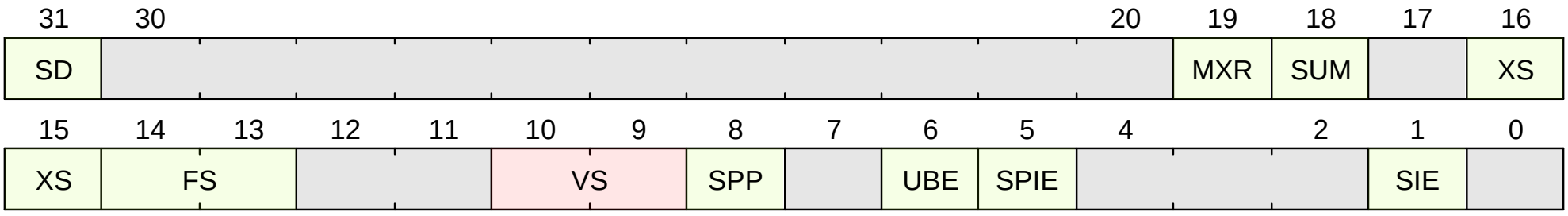


Figure 46. sstatus Format when CSR[mstatus].SXL == 0

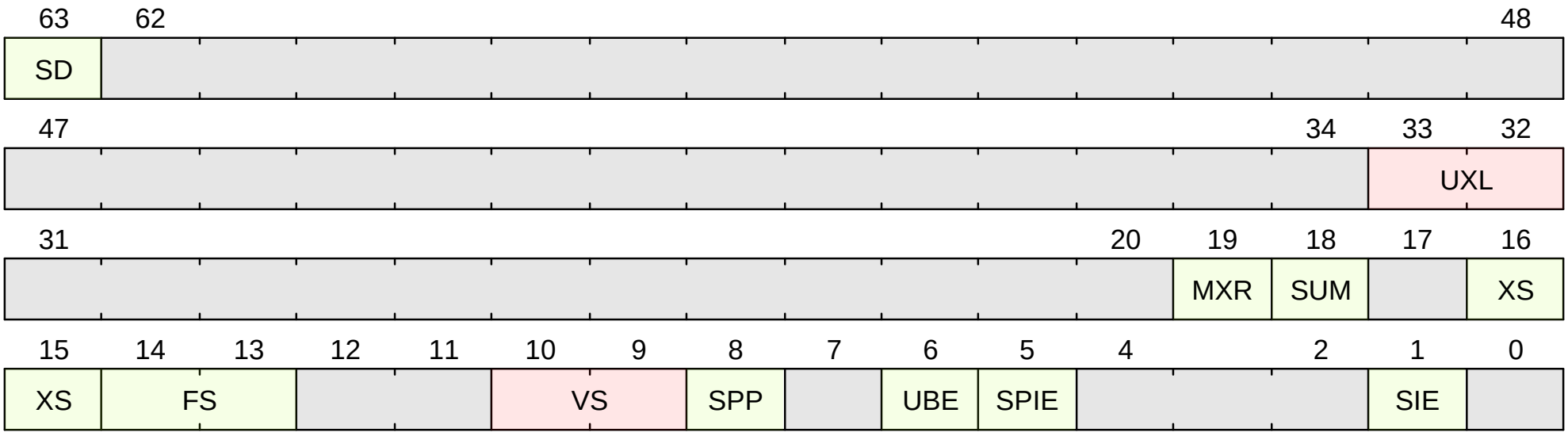


Figure 47. sstatus Format when CSR[mstatus].SXL == 1

D.44.3. Field Summary

Name	Location	Type	Reset Value
sstatus.SD	31 63	RO-H	UNDEFINED_LEGAL
sstatus.UXL	33:32	RO	2
sstatus.MXR	19	RW	UNDEFINED_LEGAL
sstatus.SUM	18	RW	UNDEFINED_LEGAL
sstatus.XS	16:15	RO	UNDEFINED_LEGAL
sstatus.FS	14:13	RW-H	UNDEFINED_LEGAL

Name	Location	Type	Reset Value
sstatus.VS	10:9	RW-H	UNDEFINED_LEGAL
sstatus.SPP	8	RW-H	UNDEFINED_LEGAL
sstatus.UBE	6	RO	UNDEFINED_LEGAL
sstatus.SPIE	5	RW-H	UNDEFINED_LEGAL
sstatus.SIE	1	RW-H	UNDEFINED_LEGAL

D.44.4. Fields

sstatus.SD Field

Location:

- 31 when CSR[mstatus].SXL == 0
- 63 when CSR[mstatus].SXL == 1

Description:

State Dirty

Alias of mstatus.SD.

Type:

RO-H

Reset value:

UNDEFINED_LEGAL

sstatus.UXL Field

UXL is only defined in RV64 (`CSR[mstatus].SXL == 1`)

Location:

33:32

Description:

U-mode XLEN

Alias of mstatus.UXL.

Type:

RO

Reset value:

2

sstatus.MXR Field

Location:

19

Description:

Make eXecutable Readable

Alias of mstatus.MXR.

Type:

RW

Reset value:

UNDEFINED_LEGAL

sstatus.SUM Field

Location:

18

Description:

permit Supervisor Memory Access

Alias of mstatus.SUM.

Type:

RW

Reset value:

UNDEFINED_LEGAL

sstatus.XS Field

Location:

16:15

Description:

Custom (X) extension context Status.

Alias of mstatus.XS.

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.FS Field

Location:

14:13

Description:

Floating point context status.

Alias of mstatus.FS.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.VS Field

Location:

10:9

Description:

Vector context status.

Alias of mstatus.VS.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.SPP Field

Location:

8

Description:

S-mode Previous Privilege

Alias of mstatus.SPP.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.UBE Field

Location:

6

Description:

U-mode Big Endian

Alias of mstatus.UBE.

Type:

RO

Reset value:

UNDEFINED_LEGAL

sstatus.SPIE Field

Location:

5

Description:

S-mode Previous Interrupt Enable

Alias of mstatus.SPIE.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

sstatus.SIE Field

Location:

1

Description:

S-mode Interrupt Enable

Alias of mstatus.SIE.

Type:

RW-H

Reset value:

UNDEFINED_LEGAL

D.45. stval

Supervisor Trap Value

Holds trap-specific information

D.45.1. Attributes

CSR Address	0x143
Defining extension	S
Length	64-bit
Privilege Mode	S

D.45.2. Format

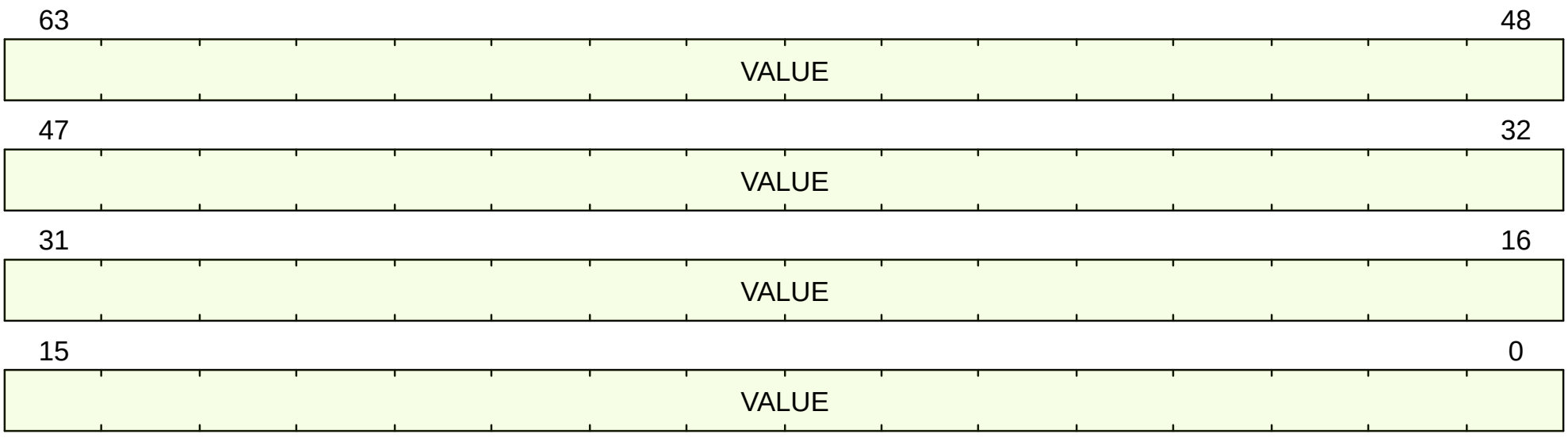


Figure 48. stval format

D.45.3. Field Summary

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

D.45.4. Fields

stval.VALUE Field

Location:

63:0

Description:

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

The values are:

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

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

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

When caused by a data address (*i.e.*, watchpoint) breakpoint, the faulting virtual address.

When caused by an instruction address breakpoint, the faulting virtual PC.
! [4] Load address misaligned ! The misaligned virtual load address.
! [5] Load access fault
! The part of virtual load address causing in the access fault.

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

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

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

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

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

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

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

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

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

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

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

Type:

RW-H

Reset value:

0

D.46. stvec

Supervisor Trap Vector

Controls where traps jump.

D.46.1. Attributes

CSR Address	0x105
Defining extension	S
Length	64-bit
Privilege Mode	S

D.46.2. Format

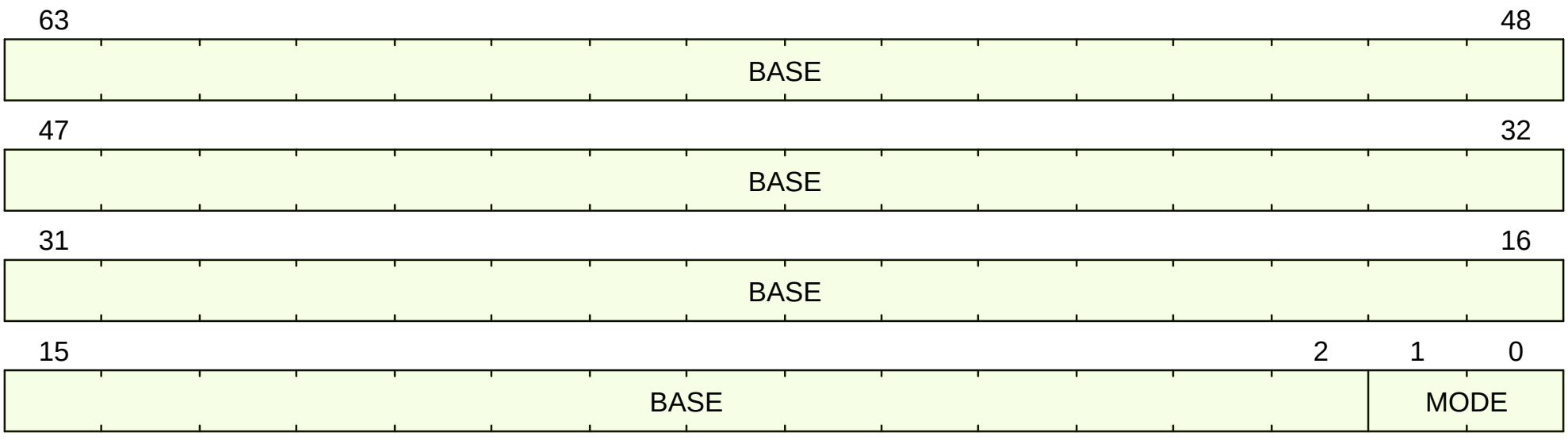


Figure 49. stvec format

D.46.3. Field Summary

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

D.46.4. Fields

stvec.BASE Field

Location:

63:2

Description:

<%- va_size = ext?(:Sv57) ? 57 : (ext?(:Sv48) ? 49 :39) -%>
Bit 63:0 of the virtual address of the exception vector for any trap taken into S-mode.

If the base address is written with a non-cannonical address (*i.e.*, bits 63:<%= va_size %> do not match bit <%= va_size-1 %>), the write should be ignored.

Type:

RW-R

Reset value:

UNDEFINED_LEGAL

stvec.MODE Field

Location:

1:0

Description:

Vectoring mode for asynchronous interrupts.

0 - Direct, 1 - Vectored

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

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

Type:

RW-R

Reset value:

0

D.46.5. Software write

This CSR may store a value that is different from what software attempts to write.

When a software write occurs (*e.g.*, through csrrw), the following determines the written value:

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

MODE = if (STVEC_MODE_DIRECT && csr_value.MODE == 0) {
  return 0;
} else if (STVEC_MODE_VECTORED && csr_value.MODE == 1) {
  return 1;
} else {
  return UNDEFINED_LEGAL_DETERMINISTIC;
}
```

D.47. 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.47.1. Attributes

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

D.47.2. Format

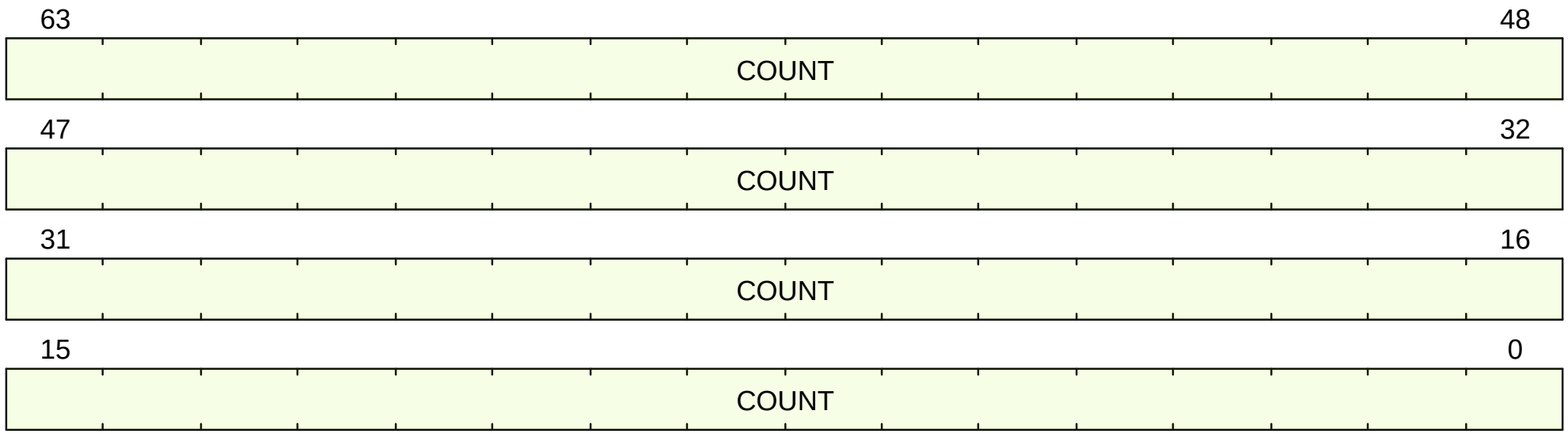


Figure 50. time format

D.47.3. Field Summary

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

D.47.4. 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.47.5. Software read

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

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

Appendix 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. unreachable (builtin)

Indicate that the IDL line should be unreachable.

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

Return Type	`void`
Arguments	None

E.11. read_hpm_counter (builtin)

Returns the value of hpmcounterN.

N must be between 3..31.

hpmcounterN must be implemented.

Return Type	`Bits`
Arguments	`Bits<5> n`

E.12. hartid (builtin)

Returns the value for mhartid as seen by this hart.

Must obey the rules of the priv spec:

> The mhartid CSR is an MXLEN-bit read-only register containing the integer ID of the hardware thread running the code. This register must be readable in any implementation. Hart IDs might not necessarily be numbered

> contiguously in a multiprocessor system, but at least one hart must have a hart ID of zero. Hart IDs must be unique within the execution environment.

Return Type	`XReg`
Arguments	None

E.13. read_mcycle (builtin)

Return the current value of the cycle counter.

Return Type	`Bits`
Arguments	None

E.14. read_mtime (builtin)

Return the current value of the real time device.

Return Type	`Bits`
Arguments	None

E.15. sw_write_mcycle (builtin)

Given a *value* that software is trying to write into mcycle, perform the write and return the value that will actually be written.

Return Type	`Bits`
Arguments	`Bits<64> value`

E.16. cache_block_zero (builtin)

Zero the cache block at the given physical address.

The cache block may be zeroed using 1 or more writes.

A cache-block-sized region is zeroed regardless of whether or not the memory is in a cacheable PMA region.

Return Type	`void`
Arguments	`XReg cache_block_physical_address`

E.17. eei_ecall_from_m (builtin)

When TRAP_ON_ECALL_FROM_M is false, this function will be called to emulate the EEI handling of ECALL-from-M.

If TRAP_ON_ECALL_FROM_M is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

E.18. eei_ecall_from_s (builtin)

When TRAP_ON_ECALL_FROM_S is false, this function will be called to emulate the EEI handling of ECALL-from-S.

If TRAP_ON_ECALL_FROM_S is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

E.19. eei_ecall_from_u (builtin)

When TRAP_ON_ECALL_FROM_U is false, this function will be called to emulate the EEI handling of ECALL-from-U.

If TRAP_ON_ECALL_FROM_U is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

E.20. eei_ecall_from_vs (builtin)

When TRAP_ON_ECALL_FROM_VS is false, this function will be called to emulate the EEI handling of ECALL-from-VS.

If TRAP_ON_ECALL_FROM_VS is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

E.21. eei_ebreak (builtin)

When TRAP_ON_EBREAK is false, this function will be called to emulate the EEI handling of EBREAK

If TRAP_ON_EBREAK is true, this function will never be called, and does not need to be provided (if pruning is applied to IDL).

Return Type	`void`
Arguments	None

E.22. memory_model_acquire (builtin)

Perform an acquire; that is, ensure that no subsequent operation in program order appears to an external observer to occur after the operation calling this function.

Return Type	`void`
Arguments	None

E.23. memory_model_release (builtin)

Perform a release; that is, ensure that no prior store in program order can be observed external to this hart after this function returns.

Return Type	`void`
Arguments	None

E.24. assert (builtin)

Assert that a condition is true. Failure represents an error in the IDL model.

Return Type	`void`
Arguments	`Boolean test, String message`

E.25. notify_mode_change (builtin)

Called whenever the privilege mode changes. Downstream tools can use this to hook events.

Return Type	`void`
Arguments	`PrivilegeMode new_mode, PrivilegeMode old_mode`

E.26. abort_current_instruction (builtin)

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

Return Type	`void`
Arguments	None

E.27. ebreak (builtin)

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

Return Type	`void`
Arguments	None

E.28. prefetch_instruction (builtin)

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

Return Type	`void`
Arguments	`XReg virtual_address`

E.29. prefetch_read (builtin)

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

Return Type	`void`
Arguments	`XReg virtual_address`

E.30. prefetch_write (builtin)

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

Return Type	`void`
Arguments	`XReg virtual_address`

E.31. fence (builtin)

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

Return Type	`void`

Arguments	`Boolean pi, Boolean pr, Boolean po, Boolean pw, Boolean si, Boolean sr, Boolean so, Boolean sw`

E.32. fence_tso (builtin)

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

Return Type	`void`
Arguments	None

E.33. ifence (builtin)

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

Return Type	`void`
Arguments	None

E.34. pause (builtin)

Pause hart retirement for a implementation-defined period of time, which may be zero.

See Zihintpause for more.

Return Type	`void`
Arguments	None

E.35. pow (generated)

Return `value` to the power `exponent`.

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

E.36. maybe_cache_translation (generated)

Given a translation result, potentially cache the result for later use. This function models a TLB fill operation. A valid implementation does nothing.

Return Type	`void`
Arguments	`XReg vaddr, MemoryOperation op, TranslationResult result`

E.37. cached_translation (generated)

Possibly returns a cached translation result matching vaddr.

CachedTranslationResult contains a Boolean 'valid' field. If valid, 'result' is a usable translation. Otherwise, the cache lookup failed.

Return Type	`CachedTranslationResult`
Arguments	`XReg vaddr, MemoryOperation op`

E.38. order_pgtbl_writes_before_vmafence (builtin)

Orders all writes prior to this call in global memory order that affect a page table in the set identified by order_type before any subsequent sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.W.INVAL.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType order_type`

E.39. order_pgtbl_reads_after_vmafence (builtin)

Orders all reads after to this call in global memory order to a page table in the set identified by order_type after any prior sfence.vma/hfence.vma/sinval.vma/hinval.gvma/hinval.vvma in program order.

Performs the ordering function of SFENCE.VMA/HFENCE.[GV]VMA/SFENCE.INVAL.IR.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType order_type`

E.40. invalidate_translations (generated)

Locally invalidate the cached S-mode/VS-mode/G-stage address translations contained in the set identified by inval_type.

A valid implementation does nothing if address caching is not used.

Return Type	`void`
Arguments	`VmaOrderType inval_type`

E.41. read_physical_memory

Read from physical memory.

Return Type	`Bits<len>`
Arguments	`XReg paddr`

```
if (len == 8) {
  return read_physical_memory_8(paddr);
} else if (len == 16) {
  return read_physical_memory_16(paddr);
} else if (len == 32) {
  return read_physical_memory_32(paddr);
} else if (len == 64) {
  return read_physical_memory_64(paddr);
} else {
  assert(false, "Invalid len");
}
```

E.42. read_physical_memory_8 (builtin)

Read a byte from physical memory.

Return Type	`Bits⑧`
Arguments	`XReg paddr`

E.43. read_physical_memory_16 (builtin)

Read two bytes from physical memory.

Return Type	`Bits⑯`
Arguments	`XReg paddr`

E.44. read_physical_memory_32 (builtin)

Read four bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

E.45. read_physical_memory_64 (builtin)

Read eight bytes from physical memory.

Return Type	`Bits`
Arguments	`XReg paddr`

E.46. write_physical_memory

Write to physical memory.

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

```
if (len == 8) {
  write_physical_memory_8(paddr, value);
} else if (len == 16) {
  write_physical_memory_16(paddr, value);
} else if (len == 32) {
  write_physical_memory_32(paddr, value);
} else if (len == 64) {
  write_physical_memory_64(paddr, value);
} else {
  assert(false, "Invalid len");
}
```

E.47. write_physical_memory_8 (builtin)

Write a byte to physical memory.

Return Type	`void`

Arguments	`XReg paddr, Bits<8> value`

E.48. write_physical_memory_16 (builtin)

Write two bytes to physical memory.

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

E.49. write_physical_memory_32 (builtin)

Write four bytes to physical memory.

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

E.50. write_physical_memory_64 (builtin)

Write eight bytes to physical memory.

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

E.51. wfi (builtin)

Wait-for-interrupt: hint that the processor should enter a low power state until the next interrupt.

A valid implementation is a no-op.

The model will advance the PC; this function does not need to.

Return Type	`void`
Arguments	None

E.52. pma_applies? (builtin)

Checks if *attr* is applied to the entire physical address region between [paddr, paddr + len) based on static PMA attributes.

Return Type	`Boolean`
Arguments	`PmaAttribute attr, Bits<PHYS_ADDR_WIDTH> paddr, U32 len`

E.53. atomic_check_then_write_32 (builtin)

Atomically:

- Reads 32-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 32-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, Bits<32> compare_value, Bits<32> write_value`

E.54. atomic_check_then_write_64 (builtin)

Atomically:

- Reads 64-bits from paddr
- Compares the read value to compare_value
- Writes write_value to paddr **if** the comparison was bitwise-equal

returns true if the write occurs, and false otherwise

Preconditions:

- paddr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, Bits<64> compare_value, Bits<64> write_value`

E.55. atomically_set_pte_a (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' bit and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len`

E.56. atomically_set_pte_a_d (builtin)

Atomically:

- Reads the *pte_len* value at *pte_addr*
  - If the read value does not exactly equal pte_value, returns false
- Sets the 'A' and 'D' bits and writes the result to *pte_addr*
- return true

Preconditions:

- pte_addr will be aligned to 64-bits

Return Type	`Boolean`

Arguments	`Bits<PHYS_ADDR_WIDTH> pte_addr, Bits<MXLEN> pte_value, U32 pte_len`

E.57. atomic_read_modify_write_32 (builtin)

Atomically read-modify-write 32-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	`Bits`
Arguments	`Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<32> value, AmoOperation op`

E.58. atomic_read_modify_write_64 (builtin)

Atomically read-modify-write 64-bits starting at phys_address using value and op.

Return the original (unmodified) read value.

All access checks/alignment checks/etc. should be done before calling this function; it's assumed the RMW is OK to proceed.

Return Type	`Bits`
Arguments	`Bits<PHYS_ADDR_WIDTH> phys_addr, Bits<64> value, AmoOperation op`

E.59. set_external_interrupt

Set an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

E.60. clear_external_interrupt

Clear an external interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

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

E.61. set_software_interrupt

Set a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

E.62. clear_software_interrupt

Clear a software interrupt targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

E.63. set_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

```
refresh_pending_interrupts();
```

E.64. clear_timer_interrupt

Set a timer interrupt from the platform targeting target_mode

Return Type	`void`
Arguments	`PrivilegeMode target_mode`

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

E.65. refresh_pending_interrupts

refreshes the calculation of a pending interrupt

needs to be called after any state update that could change a pending interrupt. This includes: - CSR[mip] - CSR[mie] - CSR[mstatus].MIE - CSR[mstatus].SIE - CSR[vsstatus].SIE - CSR[mideleg] - CSR[sideleg] - CSR[hideleg] - CSR[hvip] - CSR[hgeip] - CSR[hgeie] - mode changes

Return Type	`void`
Arguments	None

```
Bits<MXLEN> pending_ints = CSR[CSR[mip]].sw_read() & $bits(CSR[CSR[mie]]);
if (pending_ints == 0) {
  pending_and_enabled_interrupts = 0;
  return ;
}
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") &&
implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_enabled_ints = mode() == PrivilegeMode::M) && (CSR[mstatus].MIE == 1'b0 ? 0 : ($bits(CSR[CSR[mie]]) &
(HAS_MIDELEG ? ~$bits(CSR[CSR[mideleg]]) : ~MXLEN'0));
Bits<MXLEN> mmode_pending_and_enabled = pending_ints & mmode_enabled_ints;
if (mmode_pending_and_enabled != 0) {
  pending_and_enabled_interrupts = mmode_pending_and_enabled;
  return ;
}
if (CSR[misa].S == 1'b1) {
  Bits<MXLEN> smode_enabled_ints = mode() == PrivilegeMode::M) || (CSR[mstatus].SIE == 1'b0 ? 0 : $bits(CSR[CSR[mie]]) &
($bits(CSR[CSR[mideleg]]));
  Bits<MXLEN> smode_pending_and_enabled = pending_ints & smode_enabled_ints;
  if (smode_pending_and_enabled != 0) {
    pending_and_enabled_interrupts = smode_pending_and_enabled;
    return ;
  }
}
pending_and_enabled_interrupts = 0;
```

E.66. highest_priority_interrupt

Given a bitmask of interrupts in the format of MIE/MIP, return the highest priority interrupt code that is set

Interrupt priority is: MEI, MSI, MTI, SEI, SSI, STI, SGEI, VSEI, VSSI, VSTI, LCOFI

Return Type	`InterruptCode`

Arguments	Bits<MXLEN> int_mask

```
if (int_mask[$bits(InterruptCode::MachineExternal)] == 1'b1) {
  return InterruptCode::MachineExternal;
} else if (int_mask[$bits(InterruptCode::MachineSoftware)] == 1'b1) {
  return InterruptCode::MachineSoftware;
} else if (int_mask[$bits(InterruptCode::MachineTimer)] == 1'b1) {
  return InterruptCode::MachineTimer;
} else if (CSR[misa].S == 1'b1) {
  if (int_mask[$bits(InterruptCode::SupervisorExternal)] == 1'b1) {
    return InterruptCode::SupervisorExternal;
  } else if (int_mask[$bits(InterruptCode::SupervisorSoftware)] == 1'b1) {
    return InterruptCode::SupervisorSoftware;
  } else if (int_mask[$bits(InterruptCode::SupervisorTimer)] == 1'b1) {
    return InterruptCode::SupervisorTimer;
  }
} else if (implemented?(ExtensionName::Sscofpmf)) {
  if (int_mask[$bits(InterruptCode::LocalCounterOverflow)] == 1'b1) {
    return InterruptCode::LocalCounterOverflow;
  }
}
assert(false, "There is no valid interrupt");
```

E.67. choose_interrupt

Return the highest priority interrupt that is both pending and enabled and the mode it will be taken in

Return Type	InterruptCode, PrivilegeMode
Arguments	None

```
InterruptCode chosen;
Boolean HAS_MIDELEG = implemented_version?(ExtensionName::S, "<= 1.9.1") || (implemented_version?(ExtensionName::S, "> 1.9.1") &&
implemented_version?(ExtensionName::Sm, "> 1.9.1"));
Bits<MXLEN> mmode_pending_and_enabled = pending_and_enabled_interrupts & ~(HAS_MIDELEG ? $bits(CSR[CSR[mideleg]]) : MXLEN'0);
if (mmode_pending_and_enabled != 0) {
  assert((mode() != PrivilegeMode::M) || (CSR[mstatus].MIE == 1'b1), "M-mode interrupts are not enabled");
  chosen = highest_priority_interrupt(mmode_pending_and_enabled);
} else if (CSR[misa].S == 1'b1) {
  Bits<MXLEN> smode_pending_and_enabled = (pending_and_enabled_interrupts & $bits(CSR[CSR[mideleg]]));
  if (smode_pending_and_enabled != 0) {
    assert((mode() == PrivilegeMode::U) || (mode() == PrivilegeMode::VU) || (mode() == PrivilegeMode::VS) || (mode() ==
PrivilegeMode::S) && (CSR[mstatus].SIE == 1'b1), "S-mode interrupt can't be triggered");
    chosen = highest_priority_interrupt(smode_pending_and_enabled);
  }
}
assert($bits(chosen) != 0, "Didn't pick interrupt?");
PrivilegeMode to_mode;
Bits<MXLEN> chosen_mask = (MXLEN'1 << $bits(chosen));
if (((HAS_MIDELEG ? $bits(CSR[CSR[mideleg]]) : MXLEN'0) & chosen_mask) == 0) {
  to_mode = PrivilegeMode::M;
} else {
  if (CSR[misa].S == 1'b1) {
    to_mode = PrivilegeMode::S;
  } else {
    to_mode = PrivilegeMode::U;
  }
}
return chosen, to_mode;
```

E.68. take_interrupt

Take (adjust CSRs and set PC to handler) the highest priority interrupt that is both pending and enabled

Return Type	void

Arguments	None

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

E.69. fetch_memory_aligned_16

Fetch 16 bits from virtual memory using a known aligned address.

Return Type	Bits⑯
Arguments	XReg virtual_address

```
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 16, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
```

```
return read_physical_memory<16>(result.paddr);
```

E.70. fetch_memory_aligned_32

Fetch 32 bits from virtual memory using a known aligned address.

Return Type	`Bits`
Arguments	`XReg virtual_address`

```
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Fetch, mode(), virtual_address);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, 32, virtual_address, MemoryOperation::Fetch, ExceptionCode::InstructionAccessFault, mode());
return read_physical_memory<32>(result.paddr);
```

E.71. power_of_2?

Returns true if value is a power of two, false otherwise

Return Type	`Boolean`
Arguments	`Bits<N> value`

```
return (value != 0) && value & (value - 1 == 0);
```

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. max_va_size

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

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

Return the largest that applies.

Return Type	`Bits⑧`
Arguments	None

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

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

E.74. highest_set_bit

Returns the position of the highest (nearest MSB) bit that is '1', or -1 if value is zero.

Return Type	`Bits⑧`
Arguments	`XReg value`

```
for (Bits<8> i = xlen() - 1; i >= 0; i--) {
  if (value[i] == 1) {
    return i;
  }
}
return -'sd1;
```

E.75. lowest_set_bit

Returns the position of the lowest (nearest LSB) bit that is '1', or XLEN if value is zero.

Return Type	`Bits⑧`
Arguments	`XReg value`

```
for (Bits<8> i = 0; i < xlen(); i++) {
  if (value[i] == 1) {
    return i;
  }
}
return xlen();
```

E.76. bit_length

Returns the minimum number of bits needed to represent value.

Only works on unsigned values.

The value 0 returns 1.

Return Type	`XReg`
Arguments	`XReg value`

```
for (XReg i = 63; i > 0; i--) {
  if (value[i] == 1) {
    return i;
  }
}
return 1;
```

E.77. count_leading_zeros

Returns the number of leading 0 bits before the most-significant 1 bit of value, or N if value is zero.

Return Type	`Bits<bit_length(N)>`
Arguments	`Bits<N> value`

```
for (U32 i = 0; i < N; i++) {
  if (value[N - 1 - i] == 1) {
    return i;
  }
}
return N;
```

E.78. sext

Sign extend `value` starting at `first_extended_bit`.

Bits [`XLEN-1:`first_extended_bit`] of the return value should get the value of bit (`first_extended bit - 1`).

Return Type	`XReg`
Arguments	`XReg value, XReg first_extended_bit`

```
if (first_extended_bit == MXLEN) {
  return value;
} else {
  Bits<1> sign = value[first_extended_bit - 1];
  for (U32 i = MXLEN - 1; i >= first_extended_bit; i--) {
    value[i] = sign;
  }
  return value;
}
```

E.79. is_naturally_aligned

Checks if value is naturally aligned to N bits.

Return Type	`Boolean`
Arguments	`XReg value`

```
return true if (N == 8);
XReg Mask = (N / 8) - 1;
return (value & ~Mask) == value;
```

E.80. in_naturally_aligned_region?

Checks if a length-bit access starting at address lies entirely within an N-bit naturally-aligned region.

Return Type	`Boolean`
Arguments	`XReg address, U32  length`

```
XReg Mask = (N / 8) - 1;
return (address & ~Mask) == ((address + length - 1) & ~Mask);
```

E.81. contains?

Given a *region* defined by region_start, region_size, determine if a *target* defined by target_start, target_size is completely contained with the region.

Return Type	`Boolean`
Arguments	`XReg region_start, U32  region_size, XReg target_start, U32  target_size`

```
return target_start >= region_start && (target_start + target_size) <= (region_start + region_size);
```

E.82. set_fp_flag

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

Return Type	`void`
Arguments	`FpFlag flag`

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

E.83. rm_to_mode

Convert rm to a RoundingMode.

encoding is the full encoding of the instruction rm comes from.

Will raise an IllegalInstruction exception if rm is a reserved encoding.

Return Type	`RoundingMode`
Arguments	`Bits<3> rm, Bits<32> encoding`

```
if (rm == $bits(RoundingMode::RNE)) {
  return RoundingMode::RNE;
} else if (rm == $bits(RoundingMode::RTZ)) {
  return RoundingMode::RTZ;
} else if (rm == $bits(RoundingMode::RDN)) {
  return RoundingMode::RDN;
} else if (rm == $bits(RoundingMode::RUP)) {
  return RoundingMode::RUP;
} else if (rm == $bits(RoundingMode::RMM)) {
  return RoundingMode::RMM;
} else if (rm == $bits(RoundingMode::DYN)) {
  return $enum(RoundingMode, CSR[fcsr].FRM);
} else {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
}
```

E.84. mark_f_state_dirty

Potentially updates mstatus.FS to the Dirty (3) state, depending on configuration settings.

Return Type	`void`
Arguments	None

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

E.85. nan_box

Produces a properly NaN-boxed floating-point value from a floating-point value of smaller size by adding all 1's to the upper bits.

Return Type	`Bits<TO_SIZE>`
Arguments	`Bits<FROM_SIZE> from_value`

```
assert(FROM_SIZE < TO_SIZE, "Bad template arguments; FROM_SIZE must be less than TO_SIZE");
return {{TO_SIZE - FROM_SIZE{1'b1}}, from_value};
```

E.86. check_f_ok

Checks if instructions from the F extension can be executed, and, if not, raise an exception.

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

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

E.87. is_sp_neg_inf?

Return true if sp_value is negative infinity.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_NEG_INF;
```

E.88. is_sp_pos_inf?

Return true if sp_value is positive infinity.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_POS_INF;
```

E.89. is_sp_neg_norm?

Returns true if sp_value is a negative normal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 1) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

E.90. is_sp_pos_norm?

Returns true if sp_value is a positive normal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 0) && (sp_value[30:23] != 0b11111111) && !((sp_value[30:23] == 0b00000000) && sp_value[22:0] != 0);
```

E.91. is_sp_neg_subnorm?

Returns true if sp_value is a negative subnormal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 1) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

E.92. is_sp_pos_subnorm?

Returns true if sp_value is a positive subnormal number.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return (sp_value[31] == 0) && (sp_value[30:23] == 0) && (sp_value[22:0] != 0);
```

E.93. is_sp_neg_zero?

Returns true if sp_value is negative zero.

Return Type	Boolean
Arguments	Bits<32> sp_value

```
return sp_value == SP_NEG_ZERO;
```

E.94. is_sp_pos_zero?

Returns true if sp_value is positive zero.

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return sp_value == SP_POS_ZERO;
```

E.95. is_sp_nan?

Returns true if sp_value is a NaN (quiet or signaling)

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22:0] != 0);
```

E.96. is_sp_signaling_nan?

Returns true if sp_value is a signaling NaN

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 0) && (sp_value[21:0] != 0);
```

E.97. is_sp_quiet_nan?

Returns true if sp_value is a quiet NaN

Return Type	`Boolean`
Arguments	`Bits<32> sp_value`

```
return (sp_value[30:23] == 0b11111111) && (sp_value[22] == 1);
```

E.98. softfloat_shiftRightJam32

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned. The value of dist can be arbitrarily large. In particular, if dist is greater than 32, the result will be either 0 or 1, depending on whether a is zero or nonzero.

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

```
return (dist < 31) ? a >> dist | (a << (-dist & 31 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

E.99. softfloat_shiftRightJam64

Shifts a right by the number of bits given in dist, which must not be zero. If any nonzero bits are shifted off, they are "jammed" into the least-significant bit of the shifted value by setting the least-significant bit to 1. This shifted-and-jammed value is returned.

The value of 'dist' can be arbitrarily large. In particular, if dist is greater than 64, the result will be either 0 or 1, depending on whether a is zero or nonzero.

Return Type	`Bits`
Arguments	`Bits<64> a, Bits<32> dist`

```
return (dist < 63) ? a >> dist | (a << (-dist & 63 != 0) ? 1 : 0) : ((a != 0) ? 1 : 0);
```

E.100. softfloat_roundToI32

Round to signed 32-bit integer, using rounding_mode

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
    roundIncrement = 0xFFF;
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 's0) != (sign == 1) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

E.101. softfloat_roundToUI32

Round to unsigned 32-bit integer, using rounding_mode

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<64> sig, RoundingMode roundingMode`

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1) {
    if (sig == 0) {
```

```
      return 0;
    }
    if (roundingMode == RoundingMode::RDN) {
      set_fp_flag(FpFlag::NV);
    }
  } else {
    if (roundingMode == RoundingMode::RUP) {
      roundIncrement = 0xFFF;
    }
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? UI32_NEG_OVERFLOW : UI32_POS_OVERFLOW;
}
Bits<32> z = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  z = z & ~32'b1;
}
if ((z != 0) && (sign == 1)) {
  set_fp_flag(FpFlag::NV);
  return sign == 1 ? UI32_NEG_OVERFLOW : UI32_POS_OVERFLOW;
}
if (roundBits != 0) {
  set_fp_flag(FpFlag::NX);
}
return z;
```

E.102. packToF32UI

Pack components into a 32-bit value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig`

```
return {sign, exp, sig};
```

E.103. packToF16UI

Pack components into a 16-bit value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<5> exp, Bits<10> sig`

```
return {sign, exp, sig};
```

E.104. softfloat_normSubnormalF16Sig

normalize subnormal half-precision value

Return Type	`Bits<5>, Bits⑩`
Arguments	`Bits<16> hp_value`

```
Bits<8> shift_dist = count_leading_zeros<16>(hp_value);
return 1 - shift_dist, hp_value << shift_dist;
```

E.105. softfloat_roundPackToF32

Round FP value according to mdode and then pack it in IEEE format.

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
  roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
  if ($signed(exp) < 's0) {
    Boolean isTiny = ($signed(exp) < -8's1) || (sig + roundIncrement < 0x80000000);
    sig = softfloat_shiftRightJam32(sig, -exp);
    exp = 0;
    roundBits = sig & 0x7F;
    if (isTiny && (roundBits != 0)) {
      set_fp_flag(FpFlag::UF);
    }
  } else if ('shFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
    set_fp_flag(FpFlag::OF);
    set_fp_flag(FpFlag::NX);
    return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0) ? 1 : 0);   } } sig = (sig + roundIncrement);
if (sig == 0) {
  exp = 0;
}
return packToF32UI(sign, exp, sig);
```

E.106. softfloat_normRoundPackToF32

Normalize, round, and pack into a 32-bit floating point value

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
  return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
  return softfloat_roundPackToF32(sign, exp, sig << shiftDist, mode);
}
```

E.107. signF32UI

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

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

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

E.108. expF32UI

Extract exponent of a 32-bit floating point number

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

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

E.109. fracF32UI

Extract significand of a 32-bit floating point number

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

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

E.110. returnNonSignalingNaN

Returns a non-signalling NaN version of the floating-point number Does not modify the input

Return Type	`U32`
Arguments	`U32 a`

```
U32 a_copy = a;
a_copy[22] = 1'b1;
return a_copy;
```

E.111. returnMag

Returns magnitude of the given number Does not modify the input

Return Type	`U32`
Arguments	`U32 a`

```
U32 a_copy = a;
a_copy[31] = 1'b0;
return a_copy;
```

E.112. returnLargerMag

Returns the larger number between a and b by magnitude If either number is signaling NaN then that is made quiet

Return Type	`U32`
Arguments	`U32 a, U32 b`

```
U32 mag_a = returnMag(a);
U32 mag_b = returnMag(b);
U32 nonsig_a = returnNonSignalingNaN(a);
U32 nonsig_b = returnNonSignalingNaN(b);
if (mag_a < mag_b) {
```

```
  return nonsig_b;
}
if (mag_b < mag_a) {
  return nonsig_a;
}
return (nonsig_a < nonsig_b) ? nonsig_a : nonsig_b;
```

E.113. softfloat_propagateNaNF32UI

Interpreting 'a' and 'b' as the bit patterns of two 32-bit floating- | point values, at least one of which is a NaN, returns the bit pattern of | the combined NaN result. If either 'a' or 'b' has the pattern of a | signaling NaN, the invalid exception is raised.

Return Type	U32
Arguments	U32 a, U32 b

```
Boolean isSigNaN_a = is_sp_signaling_nan?(a);
Boolean isSigNaN_b = is_sp_signaling_nan?(b);
if (isSigNaN_a || isSigNaN_b) {
  set_fp_flag(FpFlag::NV);
}
return SP_CANONICAL_NAN;
```

E.114. softfloat_addMagsF32

Returns sum of the magnitudes of 2 floating point numbers

Return Type	U32
Arguments	U32 a, U32 b, RoundingMode mode

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'd0) {
    z = a + b;
    return z;
  }
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  signZ = signF32UI(a);
  expZ = expA;
  sigZ = 32'h01000000 + sigA + sigB;
  if (sigZ & 0x1) == 0) && (expZ < 8'hFE {
    sigZ = sigZ >> 1;
    return (32'h0 + (signZ << 31) + (expZ << 23) + sigZ);
  }
  sigZ = sigZ << 6;
} else {
  signZ = signF32UI(a);
  U32 sigA_32 = 32'h0 + (sigA << 6);
  U32 sigB_32 = 32'h0 + (sigA << 6);
  if (expDiff < 0) {
    if (expB == 8'hFF) {
      if (sigB != 0) {
```

```
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, 8'hFF, 23'h0);
    }
    expZ = expB;
    sigA_32 = (expA == 0) ? 2 * sigA_32 : (sigA_32 + 0x20000000);
    sigA_32 = softfloat_shiftRightJam32(sigA_32, (32'h0 - expDiff));
  } else {
    if (expA == 8'hFF) {
      if (sigA != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
    }
    expZ = expA;
    sigB_32 = (expB == 0) ? 2 * sigB_32 : (sigB_32 + 0x20000000);
    sigB_32 = softfloat_shiftRightJam32(sigB_32, (32'h0 + expDiff));
  }
  U32 sigZ = 0x20000000 + sigA + sigB;
  if (sigZ < 0x40000000) {
    expZ = expZ - 1;
    sigZ = sigZ << 1;
  }
}
return softfloat_roundPackToF32(signZ, expZ, sigZ[22:0], mode);
```

E.115. softfloat_subMagsF32

Returns difference of the magnitudes of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
Bits<8> expA = expF32UI(a);
Bits<23> sigA = fracF32UI(a);
Bits<8> expB = expF32UI(b);
Bits<23> sigB = fracF32UI(b);
U32 sigZ;
U32 z;
Bits<1> signZ;
Bits<8> expZ;
U32 sigDiff;
U32 sigX;
U32 sigY;
U32 sigA_32;
U32 sigB_32;
Bits<8> shiftDist;
Bits<8> expDiff = expA - expB;
if (expDiff == 8'd0) {
  if (expA == 8'hFF) {
    if ((sigA != 8'd0) || (sigB != 8'd0)) {
      return softfloat_propagateNaNF32UI(a, b);
    }
    return a;
  }
  sigDiff = sigA - sigB;
  if (sigDiff == 0) {
    return packToF32UI(((mode == RoundingMode::RDN) ? 1 : 0), 0, 0);
  }
  if (expA != 0) {
    expA = expA - 1;
  }
  signZ = signF32UI(a);
  if (sigDiff < 0) {
    signZ = ~signZ;
    sigDiff = -32'sh1 * sigDiff;
  }
  shiftDist = count_leading_zeros<32>(sigDiff) - 8;
  expZ = expA - shiftDist;
```

```
  if (expZ < 0) {
    shiftDist = expA;
    expZ = 0;
  }
  return packToF32UI(signZ, expZ, sigDiff << shiftDist);
} else {
  signZ = signF32UI(a);
  sigA_32 = 32'h0 + (sigA << 7);
  sigB_32 = 32'h0 + (sigB << 7);
  if (expDiff < 0) {
    signZ = ~signZ;
    if (expB == 0xFF) {
      if (sigB_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return packToF32UI(signZ, expB, 0);
    }
    expZ = expB - 1;
    sigX = sigB_32 | 0x40000000;
    sigY = sigA_32 + ((expA != 0) ? 0x40000000 : sigA_32);
    expDiff = -expDiff;
  } else {
    if (expA == 0xFF) {
      if (sigA_32 != 0) {
        return softfloat_propagateNaNF32UI(a, b);
      }
      return a;
    }
    expZ = expA - 1;
    sigX = sigA_32 | 0x40000000;
    sigY = sigB_32 + ((expB != 0) ? 0x40000000 : sigB_32);
  }
  return softfloat_normRoundPackToF32(signZ, expZ, sigX - softfloat_shiftRightJam32(sigY, expDiff), mode);
}
```

E.116. f32_add

Returns sum of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_subMagsF32(a, b, mode);
} else {
  return softfloat_addMagsF32(a, b, mode);
}
```

E.117. f32_sub

Returns difference of 2 floating point numbers

Return Type	`U32`
Arguments	`U32 a, U32 b, RoundingMode mode`

```
U32 a_xor_b = a ^ b;
if (signF32UI(a_xor_b) == 1) {
  return softfloat_addMagsF32(a, b, mode);
} else {
  return softfloat_subMagsF32(a, b, mode);
}
```

E.118. i32_to_f32

Converts 32-bit signed integer to 32-bit floating point number

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFF) == 0) {
  return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32(sign, 0x9C, magnitude_of_A, mode);
```

E.119. ui32_to_f32

Converts 32-bit unsigned integer to 32-bit floating point number

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
if (a == 0) {
  return a;
}
if (a[31] == 1) {
  return softfloat_roundPackToF32(0, 0x9D, a >> 1 | (a & 1), mode);
} else {
  return softfloat_normRoundPackToF32(0, 0x9C, a, mode);
}
```

E.120. f32_to_i32

Converts 32-bit floating point number to a signed 32-bit integer

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  return I32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToI32(sign, sig64, mode);
```

E.121. f32_to_ui32

Converts 32-bit floating point number to an unsigned 32-bit integer

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  set_fp_flag(FpFlag::NV);
  return UI32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToUI32(sign, sig64, mode);
```

E.122. softfloat_roundPackToF32_no_flag

Round FP value according to mdode and then pack it in IEEE format. No flags to be set

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> roundIncrement = 0x40;
if ((mode != RoundingMode::RNE) && (mode != RoundingMode::RMM)) {
  roundIncrement = (mode == sign != 0) ? RoundingMode::RDN : RoundingMode::RUP ? 0x7F : 0;
}
Bits<8> roundBits = sig & 0x7f;
if (0xFD <= exp) {
  if ($signed(exp) < 's0) {
    Boolean isTiny = ($signed(exp) < -8's1) || (sig + roundIncrement < 0x80000000);
    sig = softfloat_shiftRightJam32(sig, -exp);
    exp = 0;
    roundBits = sig & 0x7F;
  } else if ('shFD < $signed(exp) || (0x80000000 <= sig + roundIncrement)) {
    return packToF32UI(sign, 0xFF, 0) - roundIncrement == 0) ? 1 : 0);   } } sig = (sig + roundIncrement);
if (sig == 0) {
  exp = 0;
}
return packToF32UI(sign, exp, sig);
```

E.123. softfloat_normRoundPackToF32_no_flag

Normalize, round, and pack into a 32-bit floating point value No flags to be set

Return Type	`Bits`
Arguments	`Bits<1> sign, Bits<8> exp, Bits<23> sig, RoundingMode mode`

```
Bits<8> shiftDist = count_leading_zeros<32>(sig) - 1;
exp = exp - shiftDist;
if ((7 <= shiftDist) && (exp < 0xFD)) {
  return packToF32UI(sign, (sig != 0) ? exp : 0, sig << (shiftDist - 7));
} else {
  return softfloat_roundPackToF32_no_flag(sign, exp, sig << shiftDist, mode);
}
```

E.124. i32_to_f32_no_flag

Converts 32-bit signed integer to 32-bit floating point number No flags to be set

Return Type	U32
Arguments	U32 a, RoundingMode mode

```
Bits<1> sign = a[31];
if ((a & 0x7FFFFFFF) == 0) {
  return (sign == 1) ? packToF32UI(1, 0x9E, 0) : packToF32UI(0, 0, 0);
}
U32 magnitude_of_A = returnMag(a);
return softfloat_normRoundPackToF32_no_flag(sign, 0x9C, magnitude_of_A, mode);
```

E.125. softfloat_roundToI32_no_flag

Round to signed 32-bit integer, using rounding_mode No flag to be set

Return Type	Bits
Arguments	Bits<1> sign, Bits<64> sig, RoundingMode roundingMode

```
Bits<16> roundIncrement = 0x800;
if ((roundingMode != RoundingMode::RMM) && (roundingMode != RoundingMode::RNE)) {
  roundIncrement = 0;
  if (sign == 1 ? (roundingMode == RoundingMode::RDN) : (roundingMode == RoundingMode::RUP)) {
    roundIncrement = 0xFFF;
  }
}
Bits<16> roundBits = sig & 0xFFF;
sig = sig + roundIncrement;
if ((sig & 0xFFFFF00000000000) != 0) {
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
Bits<32> sig32 = sig >> 12;
if ((roundBits == 0x800 && (roundingMode == RoundingMode::RNE))) {
  sig32 = sig32 & ~32'b1;
}
Bits<32> z = (sign == 1) ? -sig32 : sig32;
if ((z != 0) && $signed(z) < 's0) != (sign == 1) {
  return sign == 1 ? WORD_NEG_OVERFLOW : WORD_POS_OVERFLOW;
}
return z;
```

E.126. f32_to_i32_no_flag

Converts 32-bit floating point number to a signed 32-bit integer No flags to be set

Return Type	U32
Arguments	U32 a, RoundingMode mode

```
Bits<1> sign = signF32UI(a);
Bits<8> exp = expF32UI(a);
Bits<23> sig = fracF32UI(a);
Bits<8> shiftDist;
U64 sig64;
if ((exp == 8'hFF) && (sig != 0)) {
  sign = 0;
  return I32_NAN;
}
if (exp != 0) {
  sig = sig | 32'h00800000;
}
sig64 = sig `<< 32;
shiftDist = 8'hAA - exp;
if (shiftDist > 0) {
  sig64 = softfloat_shiftRightJam64(sig64, shiftDist);
}
return softfloat_roundToI32_no_flag(sign, sig64, mode);
```

E.127. round_f32_to_integral

Rounds 32-bit floating point number to a signed 32-bit integer. This 32-bit integer is represented as a floating point number and returned.

Return Type	`U32`
Arguments	`U32 a, RoundingMode mode`

```
if ((is_sp_neg_inf?(a)) || (is_sp_pos_inf?(a)) || (is_sp_pos_zero?(a)) || (is_sp_neg_zero?(a))) {
  return a;
} else if (is_sp_signaling_nan?(a)) {
  set_fp_flag(FpFlag::NV);
  return a;
}
U32 intermediate;
intermediate = f32_to_i32_no_flag(a, mode);
return i32_to_f32_no_flag(intermediate, mode);
```

E.128. vector_state

Get the current vector state from CSRs

Return Type	`VectorState`
Arguments	None

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

E.129. mode

Returns the current active privilege mode.

Return Type	`PrivilegeMode`
Arguments	None

```
if ((!implemented?(ExtensionName::S)) && (!implemented?(ExtensionName::U)) && (!implemented?(ExtensionName::H))) {
  return PrivilegeMode::M;
} else {
  return current_mode;
}
```

E.130. set_mode_no_refresh

Set the current privilege mode to `new_mode`, but don't refresh interrupts

Return Type	`void`
Arguments	`PrivilegeMode new_mode`

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
}
```

E.131. set_mode

Set the current privilege mode to `new_mode`

Return Type	`void`
Arguments	`PrivilegeMode new_mode`

```
if (new_mode != current_mode) {
  notify_mode_change(new_mode, current_mode);
  current_mode = new_mode;
  refresh_pending_interrupts();
}
```

E.132. compatible_mode?

Returns true if target_mode is more privileged than actual_mode.

Return Type	`Boolean`
Arguments	`PrivilegeMode target_mode, PrivilegeMode actual_mode`

```
if (target_mode == PrivilegeMode::M) {
  return actual_mode == PrivilegeMode::M;
} else if (target_mode == PrivilegeMode::S) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S);
} else if (target_mode == PrivilegeMode::U) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::U);
} else if (target_mode == PrivilegeMode::VS) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS);
} else if (target_mode == PrivilegeMode::VU) {
  return (actual_mode == PrivilegeMode::M) || (actual_mode == PrivilegeMode::S) || (actual_mode == PrivilegeMode::VS) ||
(actual_mode == PrivilegeMode::VU);
}
```

E.133. exception_handling_mode

Returns the target privilege mode that will handle synchronous exception `exception_code`

Return Type	`PrivilegeMode`
Arguments	`ExceptionCode exception_code`

```
if (mode() == PrivilegeMode::M) {
  return PrivilegeMode::M;
} else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::HS) || (mode() == PrivilegeMode::U) {
  if (($bits(CSR[CSR[medeleg]]) & (MXLEN'1 << $bits(exception_code))) != 0) {
    return PrivilegeMode::HS;
  } else {
    return PrivilegeMode::M;
  }
} else {
  assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU, "Unexpected mode");
  if (($bits(CSR[CSR[medeleg]]) & (MXLEN'1 << $bits(exception_code))) != 0) {
    if (($bits(CSR[CSR[hedeleg]]) & (MXLEN'1 << $bits(exception_code))) != 0) {
      return PrivilegeMode::VS;
    } else {
      return PrivilegeMode::HS;
    }
  } else {
    return PrivilegeMode::M;
  }
}
```

E.134. creg2reg

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

Table 19. 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.135. unimplemented_csr

Either raises an IllegalInstruction exception or enters unpredictable state, depending on the setting of the TRAP_ON_UNIMPLEMENTED_CSR parameter.

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

```
if (TRAP_ON_UNIMPLEMENTED_CSR) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else {
  unpredictable("Accessing an unimplmented CSR");
```

```
}
```

E.136. mtval_readonly?

Returns whether or not CSR[mtval] is read-only based on implementation options

Return Type	`Boolean`
Arguments	None

```
return !(REPORT_VA_IN_MTVAL_ON_BREAKPOINT || REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED || REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ||
REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED || REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ||
REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT || REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT
|| REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_MTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_MTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
```

E.137. stval_readonly?

Returns whether or not CSR[stval] is read-only based on implementation options

Return Type	`Boolean`
Arguments	None

```
if (implemented?(ExtensionName::S)) {
  return !(REPORT_VA_IN_STVAL_ON_BREAKPOINT || REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED
|| REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ||
REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ||
REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT
|| REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_STVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_STVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
  return true;
}
```

E.138. vstval_readonly?

Returns whether or not CSR[vstval] is read-only based on implementation options

Return Type	`Boolean`
Arguments	None

```
if (implemented?(ExtensionName::H)) {
  return !(REPORT_VA_IN_VSTVAL_ON_BREAKPOINT || REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ||
REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ||
REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT || REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ||
REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT || REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ||
REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION || REPORT_CAUSE_IN_VSTVAL_ON_SHADOW_STACK_SOFTWARE_CHECK ||
REPORT_CAUSE_IN_VSTVAL_ON_LANDING_PAD_SOFTWARE_CHECK);
} else {
  return true;
}
```

E.139. mtval_for

Given an exception code and a **legal** non-zero value for mtval, returns the value to be written in mtval considering implementation options

Return Type	`XReg`

Arguments	ExceptionCode exception_code, XReg tval

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_MTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_MTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_MTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_MTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_MTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

E.140. stval_for

Given an exception code and a **legal** non-zero value for stval, returns the value to be written in stval considering implementation options

Return Type	XReg
Arguments	ExceptionCode exception_code, XReg tval

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_STVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_STVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

E.141. vstval_for

Given an exception code and a **legal** non-zero value for vstval, returns the value to be written in vstval considering implementation options

Return Type	`XReg`
Arguments	`ExceptionCode exception_code, XReg tval`

```
if (exception_code == ExceptionCode::Breakpoint) {
  return REPORT_VA_IN_VSTVAL_ON_BREAKPOINT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAddressMisaligned) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_MISALIGNED ? tval : 0;
} else if (exception_code == ExceptionCode::LoadAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionAccessFault) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_ACCESS_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::LoadPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_LOAD_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::StoreAmoPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_STORE_AMO_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::InstructionPageFault) {
  return REPORT_VA_IN_VSTVAL_ON_INSTRUCTION_PAGE_FAULT ? tval : 0;
} else if (exception_code == ExceptionCode::IllegalInstruction) {
  return REPORT_ENCODING_IN_VSTVAL_ON_ILLEGAL_INSTRUCTION ? tval : 0;
} else if (exception_code == ExceptionCode::SoftwareCheck) {
  return tval;
} else {
  return 0;
}
```

E.142. raise_guest_page_fault

Raise a guest page fault exception.

Return Type	`void`
Arguments	`MemoryOperation op, XReg gpa, XReg gva, XReg tinst_value, PrivilegeMode from_mode`

```
ExceptionCode code;
Boolean write_gpa_in_tval;
if (op == MemoryOperation::Read) {
  code = ExceptionCode::LoadGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_LOAD_GUEST_PAGE_FAULT;
} else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
  code = ExceptionCode::StoreAmoGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_STORE_AMO_GUEST_PAGE_FAULT;
} else {
  assert(op == MemoryOperation::Fetch, "unexpected memory operation");
  code = ExceptionCode::InstructionGuestPageFault;
  write_gpa_in_tval = REPORT_GPA_IN_TVAL_ON_INSTRUCTION_GUEST_PAGE_FAULT;
}
PrivilegeMode handling_mode = exception_handling_mode(code);
if (handling_mode == PrivilegeMode::S) {
  CSR[htval].VALUE = write_gpa_in_tval ? (gpa >> 2) : 0;
  CSR[htinst].VALUE = tinst_value;
  CSR[sepc].PC = $pc;
  if (!stval_readonly?()) {
    CSR[stval].VALUE = stval_for(code, gva);
  }
  $pc = {CSR[stvec].BASE, 2'b00};
```

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

E.143. raise

Raise synchronous exception number `exception_code`.

The exception may be imprecise, and will cause execution to enter an unpredictable state, if PRECISE_SYNCHRONOUS_EXCEPTIONS is false.

Otherwise, the exception will be precise.

Return Type	`void`
Arguments	`ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval`

```
if (!PRECISE_SYNCHRONOUS_EXCEPTIONS) {
  unpredictable("Imprecise synchronous exception");
} else {
  raise_precise(exception_code, from_mode, tval);
}
```

E.144. raise_precise

Raise synchronous exception number `exception_code`.

Return Type	`void`
Arguments	`ExceptionCode exception_code, PrivilegeMode from_mode, XReg tval`

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

```
        CSR[mstatush].MPV = 0;
      } else {
        CSR[mstatus].MPV = 0;
      }
    }
  }
  CSR[mstatus].MPP = $bits(from_mode);
} else if (CSR[misa].S == 1 && (handling_mode == PrivilegeMode::S)) {
  CSR[sepc].PC = $pc;
  if (!stval_readonly?()) {
    CSR[stval].VALUE = stval_for(exception_code, tval);
  }
  $pc = {CSR[stvec].BASE, 2'b00};
  CSR[scause].INT = 1'b0;
  CSR[scause].CODE = $bits(exception_code);
  CSR[mstatus].SPP = $bits(from_mode)[0];
  if (CSR[misa].H == 1) {
    CSR[htval].VALUE = 0;
    CSR[htinst].VALUE = 0;
    CSR[hstatus].SPV = $bits(from_mode)[2];
    if (from_mode == PrivilegeMode::VU || from_mode == PrivilegeMode::VS) {
      CSR[hstatus].SPV = 1;
      if (exception_code == ExceptionCode::Breakpoint) && (REPORT_VA_IN_STVAL_ON_BREAKPOINT || exception_code ==
ExceptionCode::LoadAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_LOAD_MISALIGNED || exception_code ==
ExceptionCode::StoreAmoAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_MISALIGNED || exception_code ==
ExceptionCode::InstructionAddressMisaligned) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_MISALIGNED || exception_code ==
ExceptionCode::LoadAccessFault) && (REPORT_VA_IN_STVAL_ON_LOAD_ACCESS_FAULT || exception_code ==
ExceptionCode::StoreAmoAccessFault) && (REPORT_VA_IN_STVAL_ON_STORE_AMO_ACCESS_FAULT || exception_code ==
ExceptionCode::InstructionAccessFault) && (REPORT_VA_IN_STVAL_ON_INSTRUCTION_ACCESS_FAULT || exception_code ==
ExceptionCode::LoadPageFault) && (REPORT_VA_IN_STVAL_ON_LOAD_PAGE_FAULT || exception_code == ExceptionCode::StoreAmoPageFault) &&
(REPORT_VA_IN_STVAL_ON_STORE_AMO_PAGE_FAULT || exception_code == ExceptionCode::InstructionPageFault) &&
(REPORT_VA_IN_STVAL_ON_INSTRUCTION_PAGE_FAULT) {
        CSR[hstatus].GVA = 1;
      } else {
        CSR[hstatus].GVA = 0;
      }
      CSR[hstatus].SPVP = $bits(from_mode)[0];
    } else {
      CSR[hstatus].SPV = 0;
      CSR[hstatus].GVA = 0;
    }
  }
} else if (CSR[misa].H == 1 && (handling_mode == PrivilegeMode::VS)) {
  CSR[vsepc].PC = $pc;
  if (!vstval_readonly?()) {
    CSR[vstval].VALUE = vstval_for(exception_code, tval);
  }
  $pc = {CSR[vstvec].BASE, 2'b00};
  CSR[vscause].INT = 1'b0;
  CSR[vscause].CODE = $bits(exception_code);
  CSR[vsstatus].SPP = $bits(from_mode)[0];
}
set_mode(handling_mode);
abort_current_instruction();
```

E.145. ialign

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

Return Type	Bits⑥
Arguments	None

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

E.146. jump

Jump to virtual address `target_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

Return Type	`void`
Arguments	`XReg target_addr`

```
if ((ialign() == 16) && target_addr & 0x1) != 0 {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
} else if ((ialign() == 32) && (target_addr & 0x3) != 0) {
  raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_addr);
}
$pc = target_addr;
```

E.147. jump_halfword

Jump to virtual halfword address `target_hw_addr`.

If target address is misaligned, raise a `MisalignedAddress` exception.

Return Type	`void`
Arguments	`XReg target_hw_addr`

```
assert((target_hw_addr & 0x1) == 0x0, "Expected halfword-aligned address in jump_halfword");
if (ialign() != 16) {
  if ((target_hw_addr & 0x3) != 0) {
    raise(ExceptionCode::InstructionAddressMisaligned, mode(), target_hw_addr);
  }
}
$pc = target_hw_addr;
```

E.148. valid_interrupt_code?

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

Return Type	`Boolean`
Arguments	`XReg code`

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

E.149. valid_exception_code?

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

Return Type	`Boolean`

Arguments	XReg code

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

E.150. xlen

Returns the effective XLEN for the current privilege mode.

Return Type	Bits⑧
Arguments	None

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

E.151. virtual_mode?

Returns True if the current mode is virtual (VS or VU).

Return Type	`Boolean`
Arguments	None

```
return (mode() == PrivilegeMode::VS) || (mode() == PrivilegeMode::VU);
```

E.152. mask_eaddr

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

Return Type	`XReg`
Arguments	`XReg eaddr`

```
return eaddr;
```

E.153. pmp_match_64

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 8) * 2;
  Bits<6> shamt = (i % 8) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<64> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_base = 0;
  Bits<PHYS_ADDR_WIDTH> range_limit = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_base = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_base = (csr_sw_read(tor_pmpaddr_csr))[PHYS_ADDR_WIDTH - 1:0];
    }
    range_limit = (pmpaddr_csr_value)[PHYS_ADDR_WIDTH - 1:0] - 1;
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 1> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 1:0];
    Bits<PHYS_ADDR_WIDTH - 1> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_base = (pmpaddr_value & ~mask);
    range_limit = range_base + mask;
  } else if (cfg.A == $bits(PmpCfg_A::NA4)) {
    range_base = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 1:0];
    range_limit = range_base + 3;
  }
  if (paddr {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
```

```
}
return PmpMatchResult::NoMatch, -;
```

E.154. pmp_match_32

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
Bits<12> pmpcfg0_addr = 0x3a0;
Bits<12> pmpaddr0_addr = 0x3b0;
for (U32 i = 0; i < NUM_PMP_ENTRIES; i++) {
  Bits<12> pmpcfg_idx = pmpcfg0_addr + (i / 4);
  Bits<6> shamt = (i % 4) * 8;
  Csr pmpcfg_csr = direct_csr_lookup(pmpcfg_idx);
  PmpCfg cfg = (csr_hw_read(pmpcfg_csr) >> shamt)[7:0];
  Bits<12> pmpaddr_idx = pmpaddr0_addr + i;
  Csr pmpaddr_csr = direct_csr_lookup(pmpaddr_idx);
  Bits<32> pmpaddr_csr_value = csr_sw_read(pmpaddr_csr);
  Bits<PHYS_ADDR_WIDTH> range_base = 0;
  Bits<PHYS_ADDR_WIDTH> range_limit = 0;
  if (cfg.A == $bits(PmpCfg_A::TOR)) {
    if (i == 0) {
      range_base = 0;
    } else {
      Csr tor_pmpaddr_csr = direct_csr_lookup(pmpaddr_idx - 1);
      range_base = csr_sw_read(tor_pmpaddr_csr)[PHYS_ADDR_WIDTH - 1:0];
    }
    range_limit = (pmpaddr_csr_value)[PHYS_ADDR_WIDTH - 1:0] - 1;
  } else if (cfg.A == $bits(PmpCfg_A::NAPOT)) {
    Bits<PHYS_ADDR_WIDTH - 1> pmpaddr_value = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 3:0];
    Bits<PHYS_ADDR_WIDTH - 1> mask = pmpaddr_value ^ (pmpaddr_value + 1);
    range_base = pmpaddr_value & ~mask;
    range_limit = range_base + mask;
  } else if (cfg.A == $bits(PmpCfg_A::NA4)) {
    range_base = pmpaddr_csr_value[PHYS_ADDR_WIDTH - 1:0];
    range_limit = range_base + 3;
  }
  if (paddr {
    return PmpMatchResult::FullMatch, cfg;
  } else if (! {
    return PmpMatchResult::PartialMatch, -;
  }
}
return PmpMatchResult::NoMatch, -;
```

E.155. pmp_match

Given a physical address, see if any PMP entry matches.

If there is a complete match, return the PmpCfg that guards the region. If there is no match or a partial match, report that result.

Return Type	`PmpMatchResult, PmpCfg`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size`

```
if (MXLEN == 64) {
  return pmp_match_64(paddr, access_size);
} else {
  return pmp_match_32(paddr, access_size);
}
```

E.156. mpv

Returns the current value of CSR[mstatus].MPV (when MXLEN == 64) of CSR[mstatush].MPV (when MXLEN == 32)

Return Type	`Bits①`
Arguments	None

```
if (implemented?(ExtensionName::H)) {
  return (MXLEN == 32) ? CSR[mstatush].MPV : CSR[mstatus].MPV;
} else {
  assert(false, "TODO");
}
```

E.157. effective_ldst_mode

Returns the effective privilege mode for normal explicit loads and stores, taking into account the current actual privilege mode and modifications from mstatus.MPRV.

Return Type	`PrivilegeMode`
Arguments	None

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

E.158. pmp_check

Given a physical address and operation type, return whether or not the access is allowed by PMP.

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, MemoryOperation type`

```
PrivilegeMode mode = effective_ldst_mode();
PmpMatchResult match_result;
PmpCfg cfg;
(match_result, cfg = pmp_match(paddr, access_size));
if (match_result == PmpMatchResult::FullMatch) {
  if (mode == PrivilegeMode::M && (cfg.L == 0)) {
    return true;
  }
  if (type == MemoryOperation::Write && (cfg.W == 0)) {
    return false;
  } else if (type == MemoryOperation::Read && (cfg.R == 0)) {
    return false;
  } else if (type == MemoryOperation::Fetch && (cfg.X == 0)) {
    return false;
  }
```

```
} else if (match_result == PmpMatchResult::NoMatch) {
  if (mode == PrivilegeMode::M) {
    return true;
  } else {
    return false;
  }
} else {
  assert(match_result == PmpMatchResult::PartialMatch, "PMP matching logic error");
  return false;
}
return true;
```

E.159. access_check

Checks if the physical address paddr is able to access memory, and raises the appropriate exception if not.

Return Type	`void`
Arguments	`Bits<PHYS_ADDR_WIDTH> paddr, U32 access_size, XReg vaddr, MemoryOperation type, ExceptionCode fault_type, PrivilegeMode from_mode`

```
if (paddr > 1 `<< PHYS_ADDR_WIDTH) - access_size {
  raise(fault_type, from_mode, vaddr);
}
if (implemented?(ExtensionName::Smpmp)) {
  if (!pmp_check(paddr[PHYS_ADDR_WIDTH - 1:0], access_size, type)) {
    raise(fault_type, from_mode, vaddr);
  }
}
```

E.160. base32?

return True iff current effective XLEN == 32

Return Type	`Boolean`
Arguments	None

```
if (MXLEN == 32) {
  return true;
} else {
  XRegWidth xlen32 = XRegWidth::XLEN32;
  if (mode() == PrivilegeMode::M) {
    return CSR[misa].MXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::S) && mode() == PrivilegeMode::S) {
    return CSR[mstatus].SXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::U) && mode() == PrivilegeMode::U) {
    return CSR[mstatus].UXL == $bits(xlen32);
  } else if (implemented?(ExtensionName::H) && mode() == PrivilegeMode::VS) {
    return CSR[hstatus].VSXL == $bits(xlen32);
  } else {
    assert(implemented?(ExtensionName::H) && mode() == PrivilegeMode::VU, "Unexpected mode");
    return CSR[vsstatus].UXL == $bits(xlen32);
  }
}
```

E.161. base64?

return True iff current effective XLEN == 64

Return Type	`Boolean`
Arguments	None

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

E.162. current_translation_mode

Returns the current first-stage translation mode for an explicit load or store from mode given the machine state (e.g., value of satp or vsatp csr).

Returns SatpMode::Reserved if the setting found in satp or vsatp is invalid.

Return Type	`SatpMode`
Arguments	`PrivilegeMode mode`

```
PrivilegeMode effective_mode = effective_ldst_mode();
if (effective_mode == PrivilegeMode::M) {
  return SatpMode::Bare;
}
if (CSR[misa].H == 1'b1) {
  if (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU) {
    Bits<4> mode_val = CSR[vsatp].MODE;
    if (mode_val == $bits(SatpMode::Bare)) {
      return SatpMode::Bare;
    } else if (mode_val == $bits(SatpMode::Sv32)) {
      if (MXLEN == 64) {
        if ((effective_mode == PrivilegeMode::VS) && (CSR[hstatus].VSXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
        if ((effective_mode == PrivilegeMode::VU) && (CSR[vsstatus].UXL != $bits(XRegWidth::XLEN32))) {
          return SatpMode::Reserved;
        }
      }
      if (!SV32_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv32;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
      if (effective_mode == PrivilegeMode::VS && CSR[hstatus].VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[vsstatus].UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV39_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv39;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
      if (effective_mode == PrivilegeMode::VS && CSR[hstatus].VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[vsstatus].UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV48_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv48;
    } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
      if (effective_mode == PrivilegeMode::VS && CSR[hstatus].VSXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::VU && CSR[vsstatus].UXL != $bits(XRegWidth::XLEN64)) {
        return SatpMode::Reserved;
      }
      if (!SV57_VSMODE_TRANSLATION) {
        return SatpMode::Reserved;
      }
      return SatpMode::Sv57;
    } else {
      return SatpMode::Reserved;
```

```
    }
  } else {
    return SatpMode::Reserved;
  }
} else if (CSR[misa].S == 1'b1) {
  assert(effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U, "unexpected priv mode");
  Bits<4> mode_val = CSR[satp].MODE;
  if (mode_val == $bits(SatpMode::Bare)) {
    return SatpMode::Bare;
  } else if (mode_val == $bits(SatpMode::Sv32)) {
    if (MXLEN == 64) {
      if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
      if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits(XRegWidth::XLEN32)) {
        return SatpMode::Reserved;
      }
    }
    if (!implemented?(ExtensionName::Sv32)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv32;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv39))) {
    if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv39)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv39;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv48))) {
    if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv48)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv48;
  } else if ((MXLEN == 64) && (mode_val == $bits(SatpMode::Sv57))) {
    if (effective_mode == PrivilegeMode::S && CSR[mstatus].SXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (effective_mode == PrivilegeMode::U && CSR[sstatus].UXL != $bits(XRegWidth::XLEN64)) {
      return SatpMode::Reserved;
    }
    if (!implemented?(ExtensionName::Sv57)) {
      return SatpMode::Reserved;
    }
    return SatpMode::Sv57;
  } else {
    return SatpMode::Reserved;
  }
} else {
  return SatpMode::Reserved;
}
```

E.163. current_gstage_translation_mode

Returns the current second-stage translation mode for a load or store from VS-mode or VU-mode.

Return Type	`HgatpMode`
Arguments	None

```
return $enum(HgatpMode, CSR[hgatp].MODE);
```

E.164. translate_gstage

Translates a guest physical address to a physical address.

Return Type	`TranslationResult`
Arguments	`XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (effective_mode == PrivilegeMode::S || effective_mode == PrivilegeMode::U) {
  result.paddr = gpaddr;
  return result;
}
Boolean mxr = CSR[mstatus].MXR == 1;
if (GSTAGE_MODE_BARE && CSR[hgatp].MODE == $bits(HgatpMode::Bare)) {
  result.paddr = gpaddr;
  return result;
} else if (SV32X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv32x4)) {
  return gstage_page_walk<32, 34, 32, 2>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV39X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv39x4)) {
  return gstage_page_walk<39, 56, 64, 3>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV48X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv48x4)) {
  return gstage_page_walk<48, 56, 64, 4>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else if (SV57X4_TRANSLATION && CSR[hgatp].MODE == $bits(HgatpMode::Sv57x4)) {
  return gstage_page_walk<57, 56, 64, 5>(gpaddr, vaddr, op, effective_mode, false, encoding);
} else {
  if (op == MemoryOperation::Read) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  } else {
    assert(op == MemoryOperation::Fetch, "unexpected memory op");
    raise_guest_page_fault(op, gpaddr, vaddr, tinst_value_for_guest_page_fault(op, encoding, true), effective_mode);
  }
}
```

E.165. tinst_value_for_guest_page_fault

Returns the value of htinst/mtinst for a Guest Page Fault

Return Type	`XReg`
Arguments	`MemoryOperation op, Bits<INSTR_ENC_SIZE> encoding, Boolean for_final_vs_pte`

```
if (for_final_vs_pte) {
  if (op == MemoryOperation::Fetch) {
    if (TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else {
      assert(TINST_VALUE_ON_FINAL_INSTRUCTION_GUEST_PAGE_FAULT == "always pseudoinstruction", "Instruction guest page faults can
only report zero/pseudo instruction in tval");
      return 0x00002000;
    }
  } else if (op == MemoryOperation::Read) {
    if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if (($array_size(VSXLEN) == 1 && VSXLEN[0] == 32) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002000;
      } else {
        return 0x00003000;
      }
    } else if (TINST_VALUE_ON_FINAL_LOAD_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
```

```
      unpredictable("Custom value written into htinst/mtinst");
    }
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always zero") {
      return 0;
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always pseudoinstruction") {
      if (($array_size(VSXLEN) == 1 && VSXLEN[0] == 32) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
        return 0x00002020;
      } else {
        return 0x00003020;
      }
    } else if (TINST_VALUE_ON_FINAL_STORE_AMO_GUEST_PAGE_FAULT == "always transformed standard instruction") {
      return tinst_transform(encoding, 0);
    } else {
      unpredictable("Custom value written into htinst/mtinst");
    }
  }
} else {
  if (REPORT_GPA_IN_TVAL_ON_INTERMEDIATE_GUEST_PAGE_FAULT) {
    if (($array_size(VSXLEN) == 1 && VSXLEN[0] == 32) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN32)) {
      return 0x00002000;
    } else if (($array_size(VSXLEN) == 1 && VSXLEN[0] == 64) || MXLEN == 64) && (CSR[hstatus].VSXL == $bits(XRegWidth::XLEN64)) {
      return 0x00003000;
    }
  }
}
```

E.166. tinst_transform

Returns the standard transformation of an encoding for htinst/mtinst

Return Type	`Bits<INSTR_ENC_SIZE>`
Arguments	`Bits<INSTR_ENC_SIZE> encoding, Bits<5> addr_offset`

```
if (encoding[1:0] == 0b11) {
  if (encoding[6:2] == 5'b00001) {
    return {{12{1'b0}}, addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b01000) {
    return {{7{1'b0}}, encoding[24:20], addr_offset, encoding[14:12], {5{1'b0}}, encoding[6:0]};
  } else if (encoding[6:2] == 5'b01011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else if (encoding[6:2] == 5'b00011) {
    return {encoding[31:20], addr_offset, encoding[14:0]};
  } else {
    assert(false, "Bad transform");
  }
} else {
  assert(false, "TODO: compressed instruction");
}
```

E.167. transformed_standard_instruction_for_tinst

Transforms an instruction encoding for htinst.

Return Type	`Bits<INSTR_ENC_SIZE>`
Arguments	`Bits<INSTR_ENC_SIZE> original`

```
assert(false, "TODO");
return 0;
```

E.168. tinst_value

Returns the value of htinst/mtinst for the given exception code.

Return Type	`XReg`
Arguments	`ExceptionCode code, Bits<INSTR_ENC_SIZE> encoding`

```
if (code == ExceptionCode::InstructionAddressMisaligned) {
  if (TINST_VALUE_ON_INSTRUCTION_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to an InstructionAddressMisaligned exception");
  }
} else if (code == ExceptionCode::InstructionAccessFault) {
  return 0;
} else if (code == ExceptionCode::IllegalInstruction) {
  return 0;
} else if (code == ExceptionCode::Breakpoint) {
  if (TINST_VALUE_ON_BREAKPOINT == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a Breakpoint exception");
  }
} else if (code == ExceptionCode::VirtualInstruction) {
  if (TINST_VALUE_ON_VIRTUAL_INSTRUCTION == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VirtualInstruction exception");
  }
} else if (code == ExceptionCode::LoadAddressMisaligned) {
  if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_ADDRESS_MISALIGNED == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadAddressMisaligned exception");
  }
} else if (code == ExceptionCode::LoadAccessFault) {
  if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_ACCESS_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadAccessFault exception");
  }
} else if (code == ExceptionCode::StoreAmoAddressMisaligned) {
  if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_ADDRESS_MISALIGNED == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAddressMisaligned exception");
  }
} else if (code == ExceptionCode::StoreAmoAccessFault) {
  if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_ACCESS_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoAccessFault exception");
  }
} else if (code == ExceptionCode::Ucall) {
  if (TINST_VALUE_ON_UCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a UCall exception");
  }
} else if (code == ExceptionCode::Scall) {
  if (TINST_VALUE_ON_SCALL == "always zero") {
    return 0;
```

```
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a SCall exception");
  }
} else if (code == ExceptionCode::Mcall) {
  if (TINST_VALUE_ON_MCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a MCall exception");
  }
} else if (code == ExceptionCode::VScall) {
  if (TINST_VALUE_ON_VSCALL == "always zero") {
    return 0;
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a VSCall exception");
  }
} else if (code == ExceptionCode::InstructionPageFault) {
  return 0;
} else if (code == ExceptionCode::LoadPageFault) {
  if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_LOAD_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a LoadPageFault exception");
  }
} else if (code == ExceptionCode::StoreAmoPageFault) {
  if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always zero") {
    return 0;
  } else if (TINST_VALUE_ON_STORE_AMO_PAGE_FAULT == "always transformed standard instruction") {
    return transformed_standard_instruction_for_tinst(encoding);
  } else {
    unpredictable("An unpredictable value is written into tinst in response to a StoreAmoPageFault exception");
  }
} else {
  assert(false, "Unhandled exception type");
}
```

E.169. gstage_page_walk

Translate guest physical address to physical address through a page walk.

May raise a Guest Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated physical address.

Return Type	`TranslationResult`
Arguments	`XReg gpaddr, XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Boolean for_final_vs_pte, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadGuestPageFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionGuestPageFault : ExceptionCode::StoreAmoGuestPageFault);
Boolean mxr = for_final_vs_pte && (CSR[mstatus].MXR == 1);
Boolean pbmte = implemented?(ExtensionName::Svpbmt) && CSR[menvcfg].PBMTE == 1;
Boolean adue = implemented?(ExtensionName::Svadu) && CSR[menvcfg].ADUE == 1;
Bits<32> tinst = tinst_value_for_guest_page_fault(op, encoding, for_final_vs_pte);
U32 max_gpa_width = LEVELS * VPN_SIZE + 2 + 12;
if (gpaddr >> max_gpa_width != 0) {
  raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
}
ppn = CSR[hgatp].PPN;
```

```
for (U32 i = (LEVELS - 1); i >= 0; i--) {
  U32 this_vpn_size = (i == (LEVELS - 1)) ? VPN_SIZE + 2 : VPN_SIZE;
  U32 vpn = (gpaddr >> (12 + VPN_SIZE * i)) & 1 << this_vpn_size) - 1);   Bits<PA_SIZE> pte_paddr = (ppn << 12) + (vpn * (PTESIZE /
8;
  if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_paddr, PTESIZE)) {
    raise(access_fault_code, PrivilegeMode::U, vaddr);
  }
  access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
  Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_paddr);
  PteFlags pte_flags = pte[9:0];
  if ((VA_SIZE != 32) && (pte[58:54] != 0)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (!implemented?(ExtensionName::Svrsw60t59b)) {
    if ((PTESIZE >= 64) && pte[60:59] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
  }
  if (!implemented?(ExtensionName::Svnapot)) {
    if ((PTESIZE >= 64) && pte[63] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
  }
  if ((PTESIZE >= 64) && !pbmte && (pte[62:61] != 0)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if ((PTESIZE >= 64) && pbmte && (pte[62:61] == 3)) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.V == 0) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.R == 0 && pte_flags.W == 1) {
    raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
  }
  if (pte_flags.R == 1 || pte_flags.X == 1) {
    if (pte_flags.U == 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    } else if ((op == MemoryOperation::Read) || (op == MemoryOperation::ReadModifyWrite)) {
      if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
      }
    }
    if ((i > 0) && (pte[(i - 1) * VPN_SIZE:0] != 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_paddr, PTESIZE)) {
          raise(access_fault_code, PrivilegeMode::U, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_paddr, PTESIZE)) {
          raise(access_fault_code, PrivilegeMode::U, vaddr);
        }
        access_check(pte_paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
          success = atomic_check_then_write_64(pte_paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
```

```
          i = i + 1;
        } else {
          result.paddr = pte_paddr;
          if (PTESIZE >= 64) {
            result.pbmt = $enum(Pbmt, pte[62:61]);
          }
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
        raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
      }
    }
  } else {
    if (i == 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    if ((VA_SIZE != 32) && pte[63] != 0) {
      raise_guest_page_fault(op, gpaddr, vaddr, tinst, effective_mode);
    }
    ppn = pte[PA_SIZE - 3:10] << 12;
  }
}
```

E.170. stage1_page_walk

Translate virtual address to physical address through a page walk.

May raise a Page Fault if an error involving the page table structure occurs along the walk.

Implicit reads of the page table are accessed check, and may raise Access Faults. Implicit writes (updates of A/D) are also accessed checked, and may raise Access Faults

The translated address *is not* accessed checked.

Returns the translated guest physical address.

Return Type	`TranslationResult`
Arguments	`Bits<MXLEN> vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PA_SIZE> ppn;
TranslationResult result;
U32 VPN_SIZE = (LEVELS == 2) ? 10 : 9;
ExceptionCode access_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadAccessFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionAccessFault : ExceptionCode::StoreAmoAccessFault);
ExceptionCode page_fault_code = op == MemoryOperation::Read ? ExceptionCode::LoadPageFault : (op == MemoryOperation::Fetch ?
ExceptionCode::InstructionPageFault : ExceptionCode::StoreAmoPageFault);
Boolean sse = false;
Boolean adue;
if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  adue = implemented?(ExtensionName::Svadu) && CSR[henvcfg].ADUE == 1;
} else {
  adue = implemented?(ExtensionName::Svadu) && CSR[menvcfg].ADUE == 1;
}
Boolean pbmte;
if (VA_SIZE == 32) {
  pbmte = false;
} else {
  if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
    pbmte = implemented?(ExtensionName::Svpbmt) && CSR[henvcfg].PBMTE == 1;
  } else {
    pbmte = implemented?(ExtensionName::Svpbmt) && CSR[menvcfg].PBMTE == 1;
  }
```

```
}
Boolean mxr;
if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS || effective_mode == PrivilegeMode::VU)) {
  mxr = (CSR[mstatus].MXR == 1) || (CSR[vsstatus].MXR == 1);
  ppn = CSR[vsatp].PPN;
} else {
  mxr = CSR[mstatus].MXR == 1;
  ppn = CSR[satp].PPN;
}
Boolean sum;
if (CSR[misa].H == 1 && (effective_mode == PrivilegeMode::VS)) {
  sum = CSR[vsstatus].SUM == 1;
} else {
  sum = CSR[mstatus].SUM == 1;
}
if ((VA_SIZE < xlen()) && (vaddr[xlen() - 1:VA_SIZE] != {xlen() - VA_SIZE{vaddr[VA_SIZE - 1]}})) {
  raise(page_fault_code, mode(), vaddr);
}
for (U32 I = (LEVELS - 1); I >= 0; I--) {
  U32 vpn = (vaddr >> (12 + VPN_SIZE * I)) & 1 `<< VPN_SIZE) - 1);   Bits<PA_SIZE> pte_gpaddr = (ppn << 12) + (vpn * (PTESIZE / 8;
  TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Read, effective_mode, encoding);
  if (!pma_applies?(PmaAttribute::HardwarePageTableRead, pte_phys.paddr, PTESIZE)) {
    raise(access_fault_code, mode(), vaddr);
  }
  access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Read, access_fault_code, effective_mode);
  Bits<PTESIZE> pte = read_physical_memory<PTESIZE>(pte_phys.paddr);
  PteFlags pte_flags = pte[9:0];
  Boolean ss_page = (pte_flags.R == 0) && (pte_flags.W == 1) && (pte_flags.X == 0);
  if ((VA_SIZE != 32) && (pte[58:54] != 0)) {
    raise(page_fault_code, mode(), vaddr);
  }
  if (pte_flags.V == 0) {
    raise(page_fault_code, mode(), vaddr);
  }
  if (!sse) {
    if ((pte_flags.R == 0) && (pte_flags.W == 1)) {
      raise(page_fault_code, mode(), vaddr);
    }
  }
  if (pbmte) {
    if (pte[62:61] == 3) {
      raise(page_fault_code, mode(), vaddr);
    }
  } else {
    if ((PTESIZE >= 64) && (pte[62:61] != 0)) {
      raise(page_fault_code, mode(), vaddr);
    }
  }
  if (!implemented?(ExtensionName::Svrsw60t59b)) {
    if ((PTESIZE >= 64) && pte[60:59] != 0) {
      raise(page_fault_code, mode(), vaddr);
    }
  }
  if (!implemented?(ExtensionName::Svnapot)) {
    if ((PTESIZE >= 64) && (pte[63] != 0)) {
      raise(page_fault_code, mode(), vaddr);
    }
  }
  if (pte_flags.R == 1 || pte_flags.X == 1) {
    Bits<PA_SIZE> paddr_base = pte[PA_SIZE - 3:I * VPN_SIZE + 10] `<< (I * VPN_SIZE + 12);
    Bits<PA_SIZE> offset = vaddr[I * VPN_SIZE + 11:0];
    if (op == MemoryOperation::Read || op == MemoryOperation::ReadModifyWrite) {
      if (!mxr) && (pte_flags.R == 0 || mxr) && (pte_flags.X == 0 && pte_flags.R == 0) {
        raise(page_fault_code, mode(), vaddr);
      }
      if (effective_mode == PrivilegeMode::U && pte_flags.U == 0) {
        raise(page_fault_code, mode(), vaddr);
      } else if (CSR[misa].H == 1 && effective_mode == PrivilegeMode::VU && pte_flags.U == 0) {
        raise(page_fault_code, mode(), vaddr);
      } else if (effective_mode == PrivilegeMode::S && pte_flags.U == 1 && !sum) {
        raise(page_fault_code, mode(), vaddr);
      } else if (effective_mode == PrivilegeMode::VS && pte_flags.U == 1 && !sum) {
        raise(page_fault_code, mode(), vaddr);
      }
    }
```

```
    if (op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite && (pte_flags.W == 0)) {
      raise(page_fault_code, mode(), vaddr);
    } else if ((op == MemoryOperation::Fetch) && (pte_flags.X == 0)) {
      raise(page_fault_code, mode(), vaddr);
    } else if ((op == MemoryOperation::Fetch) && ss_page) {
      raise(page_fault_code, mode(), vaddr);
    }
    raise(page_fault_code, mode(), vaddr) if;
    if ((pte_flags.A == 0) || pte_flags.D == 0) && ((op == MemoryOperation::Write) || (op == MemoryOperation::ReadModifyWrite)) {
      if (adue) {
        TranslationResult pte_phys = translate_gstage(pte_gpaddr, vaddr, MemoryOperation::Write, effective_mode, encoding);
        if (!pma_applies?(PmaAttribute::RsrvEventual, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        if (!pma_applies?(PmaAttribute::HardwarePageTableWrite, pte_phys.paddr, PTESIZE)) {
          raise(access_fault_code, effective_mode, vaddr);
        }
        access_check(pte_phys.paddr, PTESIZE, vaddr, MemoryOperation::Write, access_fault_code, effective_mode);
        Boolean success;
        Bits<PTESIZE> updated_pte;
        if (pte_flags.D == 0 && (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite)) {
          updated_pte = pte | 0b11000000;
        } else {
          updated_pte = pte | 0b01000000;
        }
        if (PTESIZE == 32) {
          success = atomic_check_then_write_32(pte_phys.paddr, pte, updated_pte);
        } else if (PTESIZE == 64) {
          success = atomic_check_then_write_64(pte_phys.paddr, pte, updated_pte);
        } else {
          assert(false, "Unexpected PTESIZE");
        }
        if (!success) {
          I = I + 1;
        } else {
          TranslationResult pte_phys = translate_gstage(paddr_base + offset, vaddr, op, effective_mode, encoding);
          result.paddr = pte_phys.paddr;
          result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
          result.pte_flags = pte_flags;
          return result;
        }
      } else {
        raise(page_fault_code, mode(), vaddr);
      }
    }
    TranslationResult pte_phys = translate_gstage(paddr_base + offset, vaddr, op, effective_mode, encoding);
    result.paddr = pte_phys.paddr;
    if (PTESIZE >= 64) {
      result.pbmt = pte_phys.pbmt == Pbmt::PMA ? $enum(Pbmt, pte[62:61]) : pte_phys.pbmt;
    }
    result.pte_flags = pte_flags;
      return result;
    } else {
      if (I == 0) {
        raise(page_fault_code, mode(), vaddr);
      }
      if (pte_flags.D == 1 || pte_flags.A == 1 || pte_flags.U == 1) {
        raise(page_fault_code, mode(), vaddr);
      }
      if ((VA_SIZE != 32) && (pte[62:61] != 0)) {
        raise(page_fault_code, mode(), vaddr);
      }
      if ((VA_SIZE != 32) && pte[63] != 0) {
        raise(page_fault_code, mode(), vaddr);
      }
      ppn = pte[PA_SIZE - 3:10];
    }
  }
```

E.171. translate

Translate a virtual address for operation type op that appears to execute at effective_mode.

The translation will depend on the effective privilege mode.

May raise a Page Fault or Access Fault.

The final physical address is **not** access checked (for PMP, PMA, etc., violations). (though intermediate page table reads will be)

Return Type	`TranslationResult`
Arguments	`XReg vaddr, MemoryOperation op, PrivilegeMode effective_mode, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean cached_translation_valid;
CachedTranslationResult cached_translation_result;
cached_translation_result = cached_translation(vaddr, op);
if (cached_translation_result.valid) {
  return cached_translation_result.result;
}
TranslationResult result;
if (effective_mode == PrivilegeMode::M) {
  result.paddr = vaddr;
  return result;
}
SatpMode translation_mode = current_translation_mode(effective_mode);
if (translation_mode == SatpMode::Reserved) {
  if (op == MemoryOperation::Read) {
    raise(ExceptionCode::LoadPageFault, mode(), vaddr);
  } else if (op == MemoryOperation::Write || op == MemoryOperation::ReadModifyWrite) {
    raise(ExceptionCode::StoreAmoPageFault, mode(), vaddr);
  } else {
    assert(op == MemoryOperation::Fetch, "Unexpected memory operation");
    raise(ExceptionCode::InstructionPageFault, mode(), vaddr);
  }
}
if (translation_mode == SatpMode::Bare) {
  result.paddr = vaddr;
} else if (xlen() == 32 && translation_mode == SatpMode::Sv32) {
  result = stage1_page_walk<32, 34, 32, 2>(vaddr, op, effective_mode, encoding);
} else if (xlen() == 64 && translation_mode == SatpMode::Sv39) {
  result = stage1_page_walk<39, 56, 64, 3>(vaddr, op, effective_mode, encoding);
} else if (xlen() == 64 && translation_mode == SatpMode::Sv48) {
  result = stage1_page_walk<48, 56, 64, 4>(vaddr, op, effective_mode, encoding);
} else if (xlen() == 64 && translation_mode == SatpMode::Sv57) {
  result = stage1_page_walk<57, 56, 64, 5>(vaddr, op, effective_mode, encoding);
} else {
  assert(false, "Unexpected SatpMode");
}
maybe_cache_translation(vaddr, op, result);
return result;
```

E.172. canonical_vaddr?

Returns whether or not *vaddr* is a valid (*i.e.*, canonical) virtual address.

If pointer masking (S**pm) is enabled, then vaddr will be masked before checking the canonical address.

Return Type	`Boolean`
Arguments	`XReg vaddr`

```
if (CSR[misa].S == 1'b0) {
  return true;
}
SatpMode satp_mode;
if (virtual_mode?()) {
  satp_mode = $enum(SatpMode, CSR[vsatp].MODE);
} else {
  satp_mode = $enum(SatpMode, CSR[satp].MODE);
```

```
}
XReg eaddr = mask_eaddr(vaddr);
if (SATP_MODE_BARE && (satp_mode == SatpMode::Bare)) {
  return true;
} else if ((MXLEN == 32) && satp_mode == SatpMode::Sv32) {
  return true;
} else if ((MXLEN == 64) && satp_mode == SatpMode::Sv39) {
  return eaddr[63:39] == {25{eaddr[38]}};
} else if ((MXLEN == 64) && satp_mode == SatpMode::Sv48) {
  return eaddr[63:48] == {16{eaddr[47]}};
} else if ((MXLEN == 64) && satp_mode == SatpMode::Sv57) {
  return eaddr[63:57] == {6{eaddr[56]}};
}
```

E.173. canonical_gpaddr?

Returns whether or not gpaddr is a valid (*i.e.*, canonical) guest physical address.

Return Type	`Boolean`
Arguments	`XReg gpaddr`

```
SatpMode satp_mode = $enum(SatpMode, CSR[satp].MODE);
if (satp_mode == SatpMode::Bare) {
  return true;
} else if (satp_mode == SatpMode::Sv32) {
  return true;
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv39)) {
  return gpaddr[63:39] == {25{gpaddr[38]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv48)) {
  return gpaddr[63:48] == {16{gpaddr[47]}};
} else if ((MXLEN > 32) && (satp_mode == SatpMode::Sv57)) {
  return gpaddr[63:57] == {6{gpaddr[56]}};
}
```

E.174. misaligned_is_atomic?

Returns true if an access starting at physical_address that is N bits long is atomic.

This function takes into account any Atomicity Granule PMAs, so **it should not be used for load-reserved/store-conditional**, since those PMAs do not apply to those accesses.

Return Type	`Boolean`
Arguments	`Bits<PHYS_ADDR_WIDTH> physical_address`

```
return false if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE == 0);
if (pma_applies?(PmaAttribute::MAG16, physical_address, N) && in_naturally_aligned_region?<128>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG8, physical_address, N) && in_naturally_aligned_region?<64>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG4, physical_address, N) && in_naturally_aligned_region?<32>(physical_address, N)) {
  return true;
} else if (pma_applies?(PmaAttribute::MAG2, physical_address, N) && in_naturally_aligned_region?<16>(physical_address, N)) {
  return true;
} else {
  return false;
}
```

E.175. read_memory_aligned

Read from virtual memory using a known aligned address.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault, effective_ldst_mode());
return read_physical_memory<LEN>(result.paddr);
```

E.176. read_memory

Read from virtual memory.

Return Type	`Bits<LEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  return read_memory_aligned<LEN>(virtual_address, encoding);
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with large
atomicity granule");
  physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding).paddr
: virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
    return read_physical_memory<LEN>(physical_address);
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(),
encoding).paddr : virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::LoadAddressMisaligned, mode(), virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "sequential_bytes") {
    Bits<LEN> result = 0;
    for (U32 I = 0; I < (LEN / 8); I++) {
      result = result | (read_memory_aligned<8>(virtual_address + I, encoding) `<< (8 * I));
    }
    return result;
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior when any part
of the misaligned access causes an exception");
  }
}
```

E.177. read_memory_xlen

Read XLEN bits from memory

Return Type	`Bits<MXLEN>`

Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
if (xlen() == 32) {
  return read_memory<32>(virtual_address, encoding);
} else {
  return read_memory<64>(virtual_address, encoding);
}
```

E.178. write_memory_xlen

Read XLEN bits from memory

Return Type	`void`
Arguments	`XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
if (xlen() == 32) {
  return write_memory<32>(virtual_address, value, encoding);
} else {
  return write_memory<64>(virtual_address, value, encoding);
}
```

E.179. read_memory_xlen_aligned

Read from virtual memory XLEN (which may be runtime-determined) bits using a known aligned address.

Return Type	`Bits<MXLEN>`
Arguments	`XReg virtual_address, Bits<INSTR_ENC_SIZE> encoding`

```
TranslationResult result;
if (CSR[misa].S == 1) {
  result = translate(virtual_address, MemoryOperation::Read, effective_ldst_mode(), encoding);
} else {
  result.paddr = virtual_address;
}
access_check(result.paddr, xlen(), virtual_address, MemoryOperation::Read, ExceptionCode::LoadAccessFault, effective_ldst_mode());
if (xlen() == 32) {
  return read_physical_memory<32>(result.paddr);
} else {
  return read_physical_memory<64>(result.paddr);
}
```

E.180. invalidate_reservation_set

Invalidates any currently held reservation set.

This function may be called by the platform, independent of any actions occurring in the local hart, for any or no reason.

The platform **must** call this function if an external hart or device accesses part of this reservation set while reservation_set_valid could be true.

Return Type	`void`
Arguments	None

```
reservation_set_valid = false;
```

E.181. register_reservation_set

Register a reservation for a physical address range that subsumes [physical_address, physical_address + N).

Return Type	`void`
Arguments	`Bits<MXLEN> physical_address, Bits<MXLEN> length`

```
reservation_set_valid = true;
reservation_set_address = physical_address;
if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 64-byte region") {
  reservation_set_address = physical_address & ~MXLEN'h3f;
  reservation_set_size = 64;
} else if (LRSC_RESERVATION_STRATEGY == "reserve naturally-aligned 128-byte region") {
  reservation_set_address = physical_address & ~MXLEN'h7f;
  reservation_set_size = 128;
} else if (LRSC_RESERVATION_STRATEGY == "reserve exactly enough to cover the access") {
  reservation_set_address = physical_address;
  reservation_set_size = length;
} else if (LRSC_RESERVATION_STRATEGY == "custom") {
  unpredictable("Implementations may set reservation sets of any size, as long as they cover the reserved accessed");
} else {
  assert(false, "Unexpected LRSC_RESERVATION_STRATEGY");
}
```

E.182. load_reserved

Register a reservation for virtual_address at least N bits long and read the value from memory.

If aq is set, then also perform a memory model acquire.

If rl is set, then also perform a memory model release (software is discouraged from doing so).

This function assumes alignment checks have already occurred.

Return Type	`Bits<N>`
Arguments	`Bits<MXLEN> virtual_address, Bits<1>    aq, Bits<1>    rl, Bits<INSTR_ENC_SIZE> encoding`

```
Bits<PHYS_ADDR_WIDTH> physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Read,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::LoadAccessFault, mode(), virtual_address);
}
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
register_reservation_set(physical_address, N);
if (CSR[misa].S == 1 && LRSC_FAIL_ON_VA_SYNONYM) {
  reservation_virtual_address = virtual_address;
}
return read_memory_aligned<N>(physical_address, encoding);
```

E.183. store_conditional

Atomically check the reservation set to ensure:

- it is valid
- it covers the region addressed by this store
- the address setting the reservation set matches virtual address

If the preceding are met, perform the store and return 0. Otherwise, return 1.

Return Type	Boolean
Arguments	Bits<MXLEN> virtual_address, Bits<MXLEN> value, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Bits<PHYS_ADDR_WIDTH> physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Write,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::RsrvNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
}
access_check(physical_address, N, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (aq == 1) {
  memory_model_acquire();
}
if (rl == 1) {
  memory_model_release();
}
if (reservation_set_valid == false) {
  return false;
}
if (!contains?(reservation_set_address, reservation_set_size, physical_address, N)) {
  invalidate_reservation_set();
  return false;
}
if (LRSC_FAIL_ON_NON_EXACT_LRSC) {
  if (reservation_physical_address != physical_address || reservation_size != N) {
    invalidate_reservation_set();
    return false;
  }
}
if (LRSC_FAIL_ON_VA_SYNONYM) {
  if (reservation_virtual_address != virtual_address || reservation_size != N) {
    invalidate_reservation_set();
    return false;
  }
}
write_physical_memory<N>(physical_address, value);
return true;
```

E.184. amo

Atomically read-modify-write the location at virtual_address.

The value written to virtual_address will depend on op.

If aq is 1, then the amo also acts as a memory model acquire. If rl is 1, then the amo also acts as a memory model release.

Return Type	Bits<N>
Arguments	XReg virtual_address, Bits<N> value, AmoOperation op, Bits<1> aq, Bits<1> rl, Bits<INSTR_ENC_SIZE> encoding

```
Boolean aligned = is_naturally_aligned<N>(virtual_address);
if (!aligned && MISALIGNED_LDST_EXCEPTION_PRIORITY == "high") {
  raise(ExceptionCode::StoreAmoAddressMisaligned, mode(), virtual_address);
}
Bits<PHYS_ADDR_WIDTH> physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::ReadModifyWrite,
effective_ldst_mode(), encoding).paddr : virtual_address;
if (pma_applies?(PmaAttribute::AmoNone, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
} else if (op == AmoOperation::Add || op == AmoOperation::Max || op == AmoOperation::Maxu || op == AmoOperation::Min || op ==
AmoOperation::Minu && !pma_applies?(PmaAttribute::AmoArithmetic, physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
} else if (op == AmoOperation::And || op == AmoOperation::Or || op == AmoOperation::Xor && !pma_applies?(PmaAttribute::AmoLogical,
```

```
physical_address, N)) {
  raise(ExceptionCode::StoreAmoAccessFault, mode(), virtual_address);
} else {
  assert(pma_applies?(PmaAttribute::AmoSwap, physical_address, N) && op == AmoOperation::Swap, "Bad AMO operation");
}
if (!aligned && !misaligned_is_atomic?<N>(physical_address)) {
  raise(ExceptionCode::StoreAmoAddressMisaligned, mode(), virtual_address);
}
if (N == 32) {
  return atomic_read_modify_write_32(physical_address, value, op);
} else {
  return atomic_read_modify_write_64(physical_address, value, op);
}
```

E.185. write_memory_aligned

Write to virtual memory using a known aligned address.

Return Type	`void`
Arguments	`XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
XReg physical_address;
physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
write_physical_memory<LEN>(physical_address, value);
```

E.186. write_memory

Write to virtual memory

Return Type	`void`
Arguments	`XReg virtual_address, Bits<LEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
Boolean aligned = is_naturally_aligned<LEN>(virtual_address);
XReg physical_address;
if (aligned) {
  write_memory_aligned<LEN>(virtual_address, value, encoding);
  return ;
}
if (MISALIGNED_MAX_ATOMICITY_GRANULE_SIZE > 0) {
  assert(MISALIGNED_LDST_EXCEPTION_PRIORITY == "low", "Invalid config: can't mix low-priority misaligned exceptions with large
atomicity granule");
  physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr
: virtual_address;
  if (misaligned_is_atomic?<LEN>(physical_address)) {
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
    write_physical_memory<LEN>(physical_address, value);
    return ;
  }
}
if (!MISALIGNED_LDST) {
  if (MISALIGNED_LDST_EXCEPTION_PRIORITY == "low") {
    physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(),
encoding).paddr : virtual_address;
    access_check(physical_address, LEN, virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
  }
  raise(ExceptionCode::StoreAmoAddressMisaligned, mode(), virtual_address);
} else {
  if (MISALIGNED_SPLIT_STRATEGY == "sequential_bytes") {
    for (U32 I = 0; I < (LEN / 8); I++) {
```

```
      write_memory_aligned<8>(virtual_address + I, (value >> (8 * I))[7:0], encoding);
    }
  } else if (MISALIGNED_SPLIT_STRATEGY == "custom") {
    unpredictable("An implementation is free to break a misaligned access any way, leading to unpredictable behavior when any part
of the misaligned access causes an exception");
  }
}
```

E.187. write_memory_xlen_aligned

Write to virtual memory XLEN bits (which may be runtime determined) using a known aligned address.

Return Type	`void`
Arguments	`XReg virtual_address, Bits<MXLEN> value, Bits<INSTR_ENC_SIZE> encoding`

```
XReg physical_address;
physical_address = (CSR[misa].S == 1) ? translate(virtual_address, MemoryOperation::Write, effective_ldst_mode(), encoding).paddr :
virtual_address;
access_check(physical_address, xlen(), virtual_address, MemoryOperation::Write, ExceptionCode::StoreAmoAccessFault,
effective_ldst_mode());
if (xlen() == 32) {
  write_physical_memory<32>(physical_address, value);
} else {
  write_physical_memory<64>(physical_address, value);
}
```

E.188. mstatus_sd_has_known_reset

Returns true if the mstatus.SD bit has a defined reset value, as determined by various parameters.

Return Type	`Boolean`
Arguments	None

```
Boolean fs_has_single_value = !implemented?(ExtensionName::F || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1));
Boolean vs_has_single_value = !implemented?(ExtensionName::V || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1));
return fs_has_single_value && vs_has_single_value;
```

E.189. mstatus_sd_reset_value

Returns the reset value of mstatus.SD when known

Return Type	`Bits①`
Arguments	None

```
assert(mstatus_sd_has_known_reset(), "mstatus_sd_reset_value is only defined when mstatus_sd_has_known_reset() == true");
Bits<2> fs_value, vs_value;
if ((!implemented?(ExtensionName::F)) || ($array_size(MSTATUS_FS_LEGAL_VALUES) == 1)) {
  fs_value = (!implemented?(ExtensionName::F)) ? 0 : MSTATUS_FS_LEGAL_VALUES[0];
}
if ((!implemented?(ExtensionName::V)) || ($array_size(MSTATUS_VS_LEGAL_VALUES) == 1)) {
  fs_value = (!implemented?(ExtensionName::V)) ? 0 : MSTATUS_VS_LEGAL_VALUES[0];
}
return fs_value == 3) || (vs_value == 3 ? 1 : 0;
```

E.190. check_zcmt_enabled

If the Smstateen extension is implemented, then bit 2 in mstateen0, sstateen0, and hstateen0 is implemented. If bit 2 of a controlling `stateen0` CSR is zero, then access to the `jvt` CSR and execution of a `cm.jalt` or `cm.jt` instruction by a lower privilege level results in an illegal-instruction trap (or, if appropriate, a virtual-instruction trap).

Return Type	`void`
Arguments	`Bits<INSTR_ENC_SIZE> encoding`

```
if ((mode() != PrivilegeMode::M && implemented?(ExtensionName::Smstateen) && CSR[mstateen0].JVT == 1'b0) || (mode() ==
PrivilegeMode::U && implemented?(ExtensionName::Ssstateen) && CSR[sstateen0].JVT == 1'b0)) {
  raise(ExceptionCode::IllegalInstruction, mode(), encoding);
} else if ((mode() == PrivilegeMode::VS && implemented?(ExtensionName::Ssstateen) && CSR[hstateen0].JVT == 1'b0) || (mode() ==
PrivilegeMode::VU && implemented?(ExtensionName::Ssstateen) && (CSR[sstateen0].JVT == 1'b0 || CSR[hstateen0].JVT == 1'b0))) {
  raise(ExceptionCode::VirtualInstruction, mode(), encoding);
}
```